Intel the Microcomputer Company:
When Intel invented the microcomputer in 1971, it created the era of microcomputers. Whether used in embedded applications such as automobiles or microwave ovens, or as the CPU in personal computers or supercomputers, Intel’s microcomputers have always offered leading-edge technology. Intel continues to strive for the highest standards in memory, microcomputer components, modules and systems to give its customers the best possible competitive advantages.

EMBEDDED APPLICATIONS HANDBOOK

1990
Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

376, 386, 387, 486, 4-SITE, Above, ACE51, ACE96, ACE186, ACE196, ACE960, BITBUS, COMMputer, CREDIT, Data Pipeline, DVI, ETOX, FaxBACK, Genius, i, i³, i486, i750, i860, ICE, ICeL, ICEVIEW, iCS, iDBP, iDIS, iICE, iLBX, iMDDX, iMMX, Inboard, Insite, Intel, inteL, Intel386, inteLBOS, Intel Certified, Inteltelevision, inteILient Identifier, inteILient Programming, Inteltec, Intellink, iOSP, iPAT, IPDS, IPSC, iRMK, iRMX, iSBC, isBX, iSDM, iSXl, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, PRO750, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, ToolTALK, UPI, Visual Edge, VLSICEL, and ZapCode, and the combination of ICE, iCS, iRMX, iSBC, isBX, iSXl, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

MULTIBUS is a patented Intel bus.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641
CUSTOMER SUPPORT

INTEL’S COMPLETE SUPPORT SOLUTION WORLDWIDE

Customer Support is Intel’s complete support service that provides Intel customers with hardware support, software support, customer training, consulting services and network management services. For detailed information contact your local sales offices.

After a customer purchases any system hardware or software product, service and support become major factors in determining whether that product will continue to meet a customer’s expectations. Such support requires an international support organization and a breadth of programs to meet a variety of customer needs. As you might expect, Intel’s customer support is quite extensive. It can start with assistance during your development effort to network management. 100 Intel sales and service offices are located worldwide—in the U.S., Canada, Europe and the Far East. So wherever you’re using Intel technology, our professional staff is within close reach.

HARDWARE SUPPORT SERVICES

Intel’s hardware maintenance service, starting with complete on-site installation will boost your productivity from the start and keep you running at maximum efficiency. Support for system or board level products can be tailored to match your needs, from complete on-site repair and maintenance support to economical carry-in or mail-in factory service.

Intel can provide support service for not only Intel systems and emulators, but also support for equipment in your development lab or provide service on your product to your end-user/customer.

SOFTWARE SUPPORT SERVICES

Software products are supported by our Technical Information Service (TIPS) that has a special toll free number to provide you with direct, ready information on known, documented problems and deficiencies, as well as work-arounds, patches and other solutions.

Intel’s software support consists of two levels of contracts. Standard support includes TIPS (Technical Information Phone Service), updates and subscription service (product-specific troubleshooting guides and; COMMENTS Magazine). Basic support consists of updates and the subscription service. Contracts are sold in environments which represent product groupings (e.g., iRMX® environment).

CONSULTING SERVICES

Intel provides field system engineering consulting services for any phase of your development or application effort. You can use our system engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training and customizing an Intel product to providing technical and management consulting. Systems Engineers are well versed in technical areas such as microcommunications, real-time applications, embedded microcontrollers, and network services. You know your application needs; we know our products. Working together we can help you get a successful product to market in the least possible time.

CUSTOMER TRAINING

Intel offers a wide range of instructional programs covering various aspects of system design and implementation. In just three to ten days a limited number of individuals learn more in a single workshop than in weeks of self-study. For optimum convenience, workshops are scheduled regularly at Training Centers worldwide or we can take our workshops to you for on-site instruction. Covering a wide variety of topics, Intel’s major course categories include: architecture and assembly language, programming and operating systems, BITBUS™ and LAN applications.

NETWORK MANAGEMENT SERVICES

Today’s networking products are powerful and extremely flexible. The return they can provide on your investment via increased productivity and reduced costs can be very substantial.

Intel offers complete network support, from definition of your network’s physical and functional design, to implementation, installation and maintenance. Whether installing your first network or adding to an existing one, Intel’s Networking Specialists can optimize network performance for you.
## Table of Contents

### MCS®-48 FAMILY

#### Chapter 1

**MCS®-48 APPLICATION NOTES**

- AP-24 Application Techniques for the MCS®-48 Family ........................................ 1-1
- AP-40 Keyboard/Display Scanning with Intel's MCS®-48 Microcomputers .................. 1-25
- AP-49 Serial I/O and Math Utilities for the 8049 Microcomputer ............................ 1-50
- AP-55A A High-Speed Emulator for the Intel MCS®-48 Microcomputers .................. 1-73
- AP-91 Using the 8049 as an 80 Column Printer Controller .................................... 1-173

### MCS®-51 FAMILY

#### Chapter 2

**MCS®-51 APPLICATION NOTES & ARTICLE REPRINTS**

- AP-69 An Introduction to the Intel MCS®-51 Single-Chip Microcomputer ..................... 2-1
- AP-70 Using the Intel MCS®-51 Boolean Processing Capabilities ............................. 2-31
- AP-223 8051 Based CRT Terminal Controller .......................................................... 2-76
- AB-38 Interfacing the 82786 Graphic Coprocessor to the 8051 .................................... 2-153
- AB-39 Interfacing the Densitron LCD to the 8051 ..................................................... 2-159
- AB-40 32-Bit Math Routines for the 8051 ................................................................. 2-166
- AB-12 Designing a Mailbox Memory for Two 80C31 Microcontrollers Using EPLDs .... 2-175
- AP-252 Designing with the 80C51BH ................................................................. 2-189
- AP-410 Enhanced Serial Port on the 83C51FA ......................................................... 2-213
- AB-41 Software Serial Port Implemented with the PCA ........................................... 2-221
- AP-415 83C51FA/FA PCA Cookbook ......................................................................... 2-244
- AP-425 Small DC Motor Control ............................................................................ 2-287
- AP-429 Application Techniques for the 83C152 Global Serial Channel in CSMA/CD Mode ................................................................. 2-301
- AR-517 Using the 8051 Microcontroller with Resonant Transducers ......................... 2-369
- AR-526 Analog/Digital Processing with Microcontrollers ........................................ 2-374

#### Chapter 3

**ASIC FAMILY APPLICATION NOTE & ARTICLE REPRINT**

- AP-413 Using Intel’s ASIC Core Cell to Expand the Capabilities of an 80C51 Based System ................................................................. 3-1
- AR-537 A Fast-Turnaround, Easily Testable ASIC Chip for Serial Bus Control .......... 3-11

### THE RUPI™ FAMILY

#### Chapter 4

**RUPI™ APPLICATION NOTES**

- AP-281 UPI-452 Accelerates 80286 Bus Performance ............................................ 4-1
- AP-283 RUPI™/Flexibility in Frame Size with the 8044 ........................................... 4-21

### 80186/80188 FAMILY

#### Chapter 5

**80186/80188 APPLICATION NOTES**

- AP-186 Using the 80186/188/C186/C188 Microprocessor ........................................ 5-1
- AP-258 High Speed Numerics with the 80186/80188 and 8087 .............................. 5-83
- AP-286 80186/188 Interface to Intel Microcontrollers ............................................. 5-99
- AB-36 80186/80188 DMA Latency .......................................................................... 5-129
- AB-37 80186/80188 EFI Drive and Oscillator Operation ........................................ 5-132
- AB-31 The 80C186/80C188 Integrated Refresh Control Unit .................................. 5-134
- AB-35 DRAM Refresh/Control with the 80186/188 ................................................ 5-147
Table of Contents (Continued)

MCS®-96 FAMILY

Chapter 6
MCS®-96 APPLICATION NOTES & ARTICLE REPRINT

- AP-248 Using the 8096
- AP-275 An FFT Algorithm with the MCS®-96 Products
  - Including Supporting Routines and Examples
- AB-32 Upgrade Path from 8096-90 to 8096BH to 80C196
- AB-33 Memory Expansion for the 8096
- AB-34 Integer Square Root Routine for the 8096
- AP-406 MCS®-96 Analog Acquisition Primer
- AP-428 Distributed Motor Control Using the 80C196KB
- AR-515 A Single-Chip Image Processor

Chapter 7
MCS®-96 Diagnostic Library

Chapter 8
80960 ARTICLE REPRINTS

- AB-42 80960KX Self-Test
- AR-541 Intel's 80960: An Architecture Optimized for Embedded Control
- AR-551 Embedded Controllers Push Printer Performance
- AR-557 A Programmer's View of the 80960 Architecture

GENERAL MICROCONTROLLER

Chapter 9
APPLICATION NOTES

- AP-125 Designing Microcontroller Systems for Electrically Noisy Environments
- AP-155 Oscillators for Microcontrollers
- AP-318 Intel's 87C75PF Port Expander Reduces System Size & Design Time
- AP-315 Latched EPROMs Simplify Microcontroller Designs
INTRODUCTION

The INTEL® MCS-48™ family consists of a series of seven parts, including three processors, which take advantage of the latest advances in silicon technology to provide the system designer with an effective solution to a wide variety of design problems. The significant contribution of the MCS-48 family is that instead of consisting of integrated microcomputer components it consists of integrated microcomputer systems. A single integrated circuit contains the processor, RAM, ROM (or PROM), a timer, and I/O.

This application note suggests a variety of application techniques which are useful with the MCS-48. Rather than presenting the design of a complete system it describes the implementation of “subsystems” which are common to many microprocessor based systems. The subsystems described are analog input and output, the use of tables for function evaluation, receiving serial code, transmitting serial code, and parity generation. After an overview of the MCS-48 family these areas are discussed in a more or less independent manner.

THE MCS-48™ FAMILY

The processors in the MCS-48 family all share an identical architecture. The only significant difference is the type of on board program storage which is provided. The 8748 (see Figure I) includes 1024 bytes of erasable, programmable, ROM (EPROM), the 8048 replaces the EPROM with an equivalent amount of mask programmed ROM, and the 8035 provides the CPU function with no on board program storage. All three of these processors...
### INSTRUCTION SET

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A,R</td>
<td>Add register to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD A,@R</td>
<td>Add data memory to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD A, #data</td>
<td>Add immediate to A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ADDC A,R</td>
<td>Add register with carry</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A, #data</td>
<td>Add data memory with carry</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A, @R</td>
<td>Add data memory with carry</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>AND A,R</td>
<td>And register to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AND A, #data</td>
<td>And data memory to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AND A, @R</td>
<td>And data memory to A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL A,R</td>
<td>Or register to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ORL A, #data</td>
<td>Or immediate to A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL A, @R</td>
<td>Or immediate to A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>XRL A,R</td>
<td>Exclusive or register to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>XRL A, #data</td>
<td>Exclusive or data memory to A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>XRL A, @R</td>
<td>Exclusive or immediate to A</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC A</td>
<td>Increment A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DEC A</td>
<td>Decrement A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CLR A</td>
<td>Clear A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL A</td>
<td>Complement A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DA A</td>
<td>Decimal Adjust A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SWAP A</td>
<td>Swap nibbles of A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RL A</td>
<td>Rotate A left</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate A left through carry</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RR A</td>
<td>Rotate A right</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RRC A</td>
<td>Rotate A right through carry</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN A,P</td>
<td>Input port to A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>OUTL P,A</td>
<td>Output A to port</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ANL P,#data</td>
<td>And immediate to port</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL P,#data</td>
<td>Or immediate to port</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>OUTL BUS, A</td>
<td>Output A to BUS</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ANL BUS,#data</td>
<td>And immediate to BUS</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL BUS,#data</td>
<td>Or immediate to BUS</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MOVDA P,A</td>
<td>Input Expander port to A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MOVDP A, P</td>
<td>Output A to Expander port</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ANLP A, P</td>
<td>And A to Expander port</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ORLP A, P</td>
<td>Or A to Expander port</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC R</td>
<td>Increment register</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>INC #R</td>
<td>Increment data memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DEC R</td>
<td>Decrement register</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP addr</td>
<td>Jump unconditional</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JMPP @A</td>
<td>Jump indirect</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DJNZ R, addr</td>
<td>Decrement register and skip</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JC addr</td>
<td>Jump on Carry = 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JNC addr</td>
<td>Jump on Carry = 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JZ addr</td>
<td>Jump on A Zero</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JNZ addr</td>
<td>Jump on A not Zero</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JTO addr</td>
<td>Jump on TO = 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JNTD addr</td>
<td>Jump on TO = 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JT1 addr</td>
<td>Jump on T1 = 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JNT1 addr</td>
<td>Jump on T1 = 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JF0 addr</td>
<td>Jump on F0 = 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JF1 addr</td>
<td>Jump on F1 = 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JTF addr</td>
<td>Jump on timer flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JNE addr</td>
<td>Jump on INT = 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JBb addr</td>
<td>Jump on Accumulator Bit</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

### Subroutines

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Jump to subroutine</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>RET</td>
<td>Return</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RETR</td>
<td>Return and restore status</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Flags

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR C</td>
<td>Clear Carry</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL C</td>
<td>Complement Carry</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CLR F0</td>
<td>Clear Flag 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL F0</td>
<td>Complement Flag 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CLR F1</td>
<td>Clear Flag 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL F1</td>
<td>Complement Flag 1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Data Movers

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV A,R</td>
<td>Move register to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV A,@R</td>
<td>Move data memory to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV A, #data</td>
<td>Move immediate to A</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MOV R,A</td>
<td>Move A to register</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV @R, A</td>
<td>Move A to data memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV @R, #data</td>
<td>Move immediate to data memory</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MOV A, PSW</td>
<td>Move PSW to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV PSW, A</td>
<td>Move A to PSW</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>XCH A, R</td>
<td>Exchange A and register</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>XCH A, #data</td>
<td>Exchange A and data memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>XCHD A, @R</td>
<td>Exchange nibble of A and register</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOVX A, @R</td>
<td>Move external data memory to A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOVX @R, A</td>
<td>Move A to external data memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOVPA, @A</td>
<td>Move to A from current page</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MOVPA3, @A</td>
<td>Move to A from Page 3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Timer/Counter

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV A, T</td>
<td>Read Timer/Counter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV T, A</td>
<td>Load Timer/Counter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>STRT T</td>
<td>Start Timer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>STRT CNT</td>
<td>Start Counter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>STOP TCNT</td>
<td>Stop Timer/Counter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EN TCNTI</td>
<td>Enable Timer/Counter Interrupt</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DIS TCNTI</td>
<td>Disable Timer/Counter Interrupt</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Control

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN I</td>
<td>Enable external interrupt</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DIS I</td>
<td>Disable external interrupt</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SEL RB0</td>
<td>Select register bank 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SEL RB1</td>
<td>Select register bank 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SEL MB0</td>
<td>Select memory bank 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SEL MB1</td>
<td>Select memory bank 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ENTO CLK</td>
<td>Enable Clock output on T0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Mnemonics copyright Intel Corporation 1976**

---

*Figure 2. 8048/8748/8035 Instruction Set*
operate from a single 5-volt power supply. The 8748 requires an additional 25-volt supply only while the on board EPROM is being programmed. When installed in a system only the 5-volt supply is needed. Aside from program storage, these chips include 64 bytes of data storage (RAM), an eight bit timer which can also be used to count external events, 27 programmable I/O pins and the processor itself. The processor offers a wide range of instruction capability including many designed for bit, nibble, and byte manipulation. The instruction set is summarized in Figure 2.

Aside from the processors, the MCS-48 family includes 4 devices: one pure I/O device and 3 combination memory and I/O devices. The pure I/O device is the 8243, a device which is connected to a special 4 bit bus provided by the MCS-48 processors and which provides 16 I/O pins which can be programmatically controlled.

The combination memory and I/O devices consist of the 8355, the 8755, and the 8155. The 8355 and the 8755 both provide 2,048 bytes of program storage and two eight bit data ports. The only difference between these devices is that the 8355 contains masked program ROM and the 8755 contains EPROM. The 8155 combines 256 bytes of data storage (RAM), two eight bit data ports, a six bit control port, and a 14 bit programmable timer.

Figure 3 shows the various system configurations which can be achieved using the MCS-48 family of parts. It should also be noted that eight of the processors' I/O lines have been configured as a bidirectional bus which can be used to interface to standard Intel peripheral parts such as the 8251 USART (for serial I/O), the 8255A PPI (provides 24 I/O lines) and the complete range of memory components.

More detailed information concerning the MCS-48 family can be obtained from the "MCS-48 Microcomputer User's Manual" which provides a complete description of the MCS-48 family and its members. A general familiarity with this document will make the application techniques which follow easier to understand.

### ANALOG I/O

If analog I/O is required for a MCS-48™ system there are many alternatives available from the makers of analog I/O modules. By searching through their catalogs it is possible to find almost any combination of features which is technically feasible. Perhaps the best example of such modules are the MP-10 and MP-20 hybrid modules recently introduced by Burr-Brown Research Corporation. The MP-10 provides two analog outputs and the MP-20 provides 16 analog inputs. Both of these units were specifically designed to interface with microprocessors.

A block diagram of the MP-10 is shown in Figure 4. It consists of two eight bit digital to analog converters, two eight bit latches which are loaded from the data bus, and address decoding logic to determine when the latches should be loaded. The D/A converters each generate an analog output in the range of 10 volts with an output impedance of 1Ω. Accuracy is ±0.4% of full scale and the output is stable 25μsec after the eight bit binary data is loaded into the appropriate latch. The latches are loaded by the write pulse (WR) whenever the proper address is presented to the MP-10. The lower two addresses (A0 and A1) are used internally by the device. Addresses A2 & A3 are compared with the address determination inputs B2 and B3. If their signals are found to be equal, and if addresses A4-A13 are all high, then the device is selected and one of the latches will be loaded. Address bit A1 selects between output 1 and output 2. If address bit A0 is set then the initialization channel of the DIA is selected. In order to prepare for operation a data pattern of 80H must

#### Table: Number of Available Timers and I/O Lines

<table>
<thead>
<tr>
<th>Timers</th>
<th>I/O Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-8155</td>
<td>8035</td>
</tr>
<tr>
<td>3-8155</td>
<td>8035</td>
</tr>
<tr>
<td>4-8155</td>
<td>8035</td>
</tr>
<tr>
<td>2-8355</td>
<td>8035</td>
</tr>
<tr>
<td>3-8355</td>
<td>8035</td>
</tr>
<tr>
<td>4-8355</td>
<td>8035</td>
</tr>
</tbody>
</table>

#### Figure 3. The Expanded MCS-48™ System

![Diagram showing various system configurations](image-url)
be output to this channel following the reset of the device.

A block diagram of the MP-20 analog to digital converter is shown in figure 5. This unit consists of a 16 input analog multiplexer, an instrumentation amplifier, an eight bit successive approximation analog to digital converter, and control logic. The 16 input multiplexer can be used to input either 16 single ended or 8 differential inputs. The output from the multiplexer is fed into the instrumentation amplifier which is configured so that it can easily be strapped for single ended 0-5 volt inputs, single ended ±5 volt inputs, or differential 0-5 volt signals. Provisions are made for an external gain control resistor on the amplifier. The gain control equation is:

$$G = 2 + \frac{50k\Omega}{R_{ext}}$$

With no $R_{ext}$ ($R_{ext} = \infty$) the gain is two and the input is 0-5 or ±5 volts full scale. Adding an external resistor results in higher gain so that low level (±50mV) signals from thermocouples and strain gauges can be accommodated. The output from the amplifier is applied to the actual A/D converter which provides an eight bit output with guaranteed monotonicity and an accuracy of ±0.4% of full scale. Note that this accuracy is specified for the entire module, not just for the converter itself. The control logic monitors address lines A15 through A4 to determine when the address of the unit has been selected. An address that the unit will respond to is determined by 11 address control pins, labeled A4 through A14. If one of these pins is tied to a logic 0 then the corresponding address pin must be high in order for the unit to be selected. If the pin is tied to a logic 1 then the corresponding address pin must be low. If the address of the module is selected when MEMR pulse occurs, the lower four addresses (A3-A0) are stored in a latch which addresses the multiplexer. The coincidence of the proper address and MEMR also initiates a conversion and gates the output of the converter on to the eight bit data bus.

The control logic of the MP-20 was designed to operate directly with an MCS-80™ system. When a MEMR occurs and a conversion is initiated the MP-20 generates a READY signal which is used to extend the cycle of the 8080A for the duration of the conversion. READY is brought high after the conversion is complete which allows the 8080A to initiate a conversion and read the resulting data in a single, albeit long, memory or I/O cycle. The conversion time of the MP-20 depends on the gain selected for the amplifier. With no external resistor ($R = \infty$) the gain is two and the conversion time is 35 μsec. For $R = 510\Omega$ the gain is:

$$G = 2 + \frac{50k\Omega}{51k\Omega} \approx 100$$

and the conversion time becomes 100μsec. These settling times are specified in the MP-20 data sheet and range from 35 to 175 microseconds. The READY timing is controlled by an external capacitor. For a gain of 2 no external capacitor is required but if higher gains are selected a capacitor is needed to extend the timing.

A schematic showing both the MP-10 D/A and the MP-20 A/D connected to the 8748 is shown in Figure 6. This configuration, which consists of only four major components, gives an excellent example of what modern technology can do for
MCS-48™ Based Analog Processor
the system designer. The four components provide:

a. An eight bit microprocessor
b. 64 bytes of RAM
c. 1024 bytes of UV erasable PROM
d. A timer/event counter
e. 16 digital I/O pins
f. 2 testable input pins
g. An interrupt capability
h. 16 eight bit analog inputs
i. 2 eight bit analog outputs

The MCS-48 communicates with the D/A and A/D converters in a memory mapped mode (i.e., it treats the devices as if they were external RAM). By setting an address in either R0 or R1 and then executing a MOVX the software can transfer data between the accumulator and the analog I/O. When the MCS-48 executes the MOVX instruction it first sends the eight bit address out on the bus and strobes it into the 8212 latch with the ALE (Address Latch Enable) signal. After the address is latched, the MCS-48 uses the same bus to transfer data to or from the accumulator. If data is being sent out (MOVX dRj, A) the WR strobe is used; if the data is being moved into the accumulator (MOVX A, dRj) the RD strobe is used. The one shots on the WR line are used to delay the write strobe of the 8212 latch with the ALE (Address Latch Enable) signal. After the address is latched, the MCS-48 executes the MOVX instruction it first sends the eight bit address out on the bus and strobes it into the 8212 latch with the ALE (Address Latch Enable) signal. After the address is latched, the MCS-48 uses the same bus to transfer data to or from the accumulator. If data is being sent out (MOVX dRj, A) the WR strobe is used; if the data is being moved into the accumulator (MOVX A, dRj) the RD strobe is used. The one shots on the WR line are used to delay the write strobe of the MCS-48 to meet the data set up specifications of the MP-10.

In order to provide reset capability for the analog devices without dedicating an I/O pin from the MCS-48, special addresses are used as reset channels. Executing any MOVX with an address of OXXXXXXX will reset the A/D; a similar operation with an address of X1XXXXXX will reset the D/A; a MOVX with an address of 01XXXXXX will reset both devices. All data transfers are accomplished with the upper two bits of the address field equal to 10. A summary of the addressing of the analog devices is shown in Table 1. Notice that except for an initialization channel for the D/A (which must

be written to following a reset to initialize its internal logic) all channels involve some form of data transfer.

As was mentioned previously, the MP-20 was designed to use the READY line of the 8080A. Obviously this presents a problem since the MCS-48 does not support a READY line (with its attendant requirement of entering WAIT state). The necessity of a READY input can be overcome by performing a read operation to set the channel address, waiting the required delay (35 \( \mu \text{sec} \) for a gain of two) and then performing a second read to actually obtain the data. The second read will read in the data from the channel selected by the first read irrespective of the channel selected for the second read. Thus it is possible to use the second read to set up the channel for the third read. Each read can read in the current channel and select the next channel for conversion.

The MP-20 is shown in Figure 6 strapped to input 16 single ended ±5 volts signals. Programs which were used to test this configuration are shown in Figure 7. The first of these programs uses the D/A converter to generate sawtooth waveforms by outputting an incrementing value to the D/A converters. The second program scans the analog inputs and stores their digital values in a table located in RAM.

<table>
<thead>
<tr>
<th>INPUT OR OUTPUT</th>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 XXX X XXXX</td>
<td>0 0 1 1 n n n</td>
<td>1 0 1 1 0 0 0 1</td>
</tr>
<tr>
<td>X 1 X X X XXXX</td>
<td>n n n n</td>
<td>Initialize D/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write Channel 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write Channel 2</td>
</tr>
</tbody>
</table>

Table 1. Analog Interface Addresses

LOC OBJ SEG SOURCE STATEMENT
0 0
1 0
2 0
3 0
4 0
5 0
6 0
7 0
8 0
9 0
10 0
11 0
12 0
13 0
14 0
15 0
16 0
17 0
18 0
19 0
20 0
21 0
22 0
23 0
24 0
25 0
26 0
27 0
28 0
29 0
30 0
31 0

Figure 7a. D/A Exercise Program
Where $K$ is chosen to be large enough to provide the required accuracy. As an alternative to calculating these values in real time, the values could be precalculated off line and stored in a table. Upon each interrupt the MCS-48 would merely have to retrieve the appropriate value from the table and output it to the D/A converter. The MCS-48 provides a special instruction which can be used to access data in a table. If the table is stored in the last 256 bytes of the first kilobyte of MCS-48 memory then the table lookup can be performed by loading the independent variable (time in this case) into the accumulator and executing the instruction.

\[ \text{MOVP3 A, @ A} \]

This instruction uses the initial contents of the accumulator to index into page 3 of program storage. The location pointed to is read and the contents placed in the accumulator. If (as is often the case) a table of fewer than 256 entries is required, then the table can be located in any page of program memory and the instruction:

\[ \text{MOVP A, @ A} \]

can be used to retrieve data from the table. This instruction operates in the same manner as does the previous instruction except that the current page of program memory is assumed to contain the table.

If it is possible to devote slightly more of the microprocessor's time to the table look up process, then a much smaller table can often be utilized by taking advantage of interpolation to determine values of the function between values which are actual entries in the table. As an example of this

\[ \text{Figure 7b. A/D Exercise Program} \]

\textbf{TABLE LOOKUP TECHNIQUES}

In the previous section the interface between analog I/O devices and the MCS-48 was discussed. In many applications involving analog I/O one quickly finds that nature is inherently nonlinear, and the mathematics involved in 'linearizing it' can tax the computational power of the microprocessor, particularly if it has other tasks to perform. Problems of this nature are good candidates for the use of tables.

As an example of how tables can be used as part of an analog output scheme, consider a system which requires an MCS-48 to output a variable frequency sinusoidal waveform. One method of performing this function would be to use the timer to generate an interrupt at a fixed rate of 256 times the desired output frequency. At each interrupt the appropriate value of the sine function could be calculated from the MacLaurin series:

\[ \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \]

\[ = \frac{(-1)^k x^{2k+1}}{(2K+1)!} \]

Where $K$ is chosen to be large enough to provide the required accuracy.

All mnemonics copyrighted © Intel Corporation 1976.
process consider the hypothetical system shown in Figure 8. The purpose of this system is to measure the flow through the three pipes, add them, and display the total flow on the control panel. The system consists of three flow meters which generate a differential voltage which is some function of flow, an A/D system with at least three differential inputs, an MCS48, and a control panel. The schematic shown in Figure 6 could easily become part of this system, with the spare digital I/O of the MCS48 used as an interface to the control panel. The simplicity of this system is clouded by the flow transducers, which are assumed to be not only nonlinear but also to require individual calibration (this is not an unreasonable assumption for a flow transducer). By using a table look up process and an 8748 the flow transducers can be calibrated and the results of the calibration tests stored directly in tables in the 8748. (The 8748 has a PROM in place of the ROM of the 8048 and thus makes such 'one off' programming practical.)

The results which might be obtained from calibrating one of the flow meters is shown in Figure 9. The results are plotted as gals/hour versus the measured voltage generated by the transducer. The voltage is shown in hexadecimal form so that it corresponds directly to the digital output of the analog to digital converter. The flow required to generate seventeen evenly spaced voltages (0H-10H in steps of 10H) has been measured and plotted. This information is shown in tabular form in Figure 10. It is necessary to generate a program which will convert any measured input from 00H to FFH into the flow in units which can be interpreted by a human operator. This can easily be done by simple interpolation.

The eight bits of independent variable (voltage) can be looked on as two four bit fields. The most significant four bits (7-4) will be used to retrieve one of the table values. The lower four bits (3-0) will be used to interpolate between this value and the value retrieved from the next higher location in the table. If the upper four bits are given the symbol I and the lower four bits the symbol N, then the interpolation can be expressed as:

\[ F(x) = F(I) + \frac{N}{16} [F(I+1) - F(I)] \]

Where \( x \) is the measured voltage and \( F(x) \) is the corresponding flow.

If, as an example, the transducer voltage was measured as 48H then the flow (ref. Figure 10) would be:

\[ F = 30 + \frac{8}{16} (34-30) = 32 \]

A subroutine which implements this calculation is shown in Figure 11. Before it is called the independent variable (V) is placed in the accumulator and register R1 is set to point at the first value in the table. Aside from simple additions and subtractions the only arithmetic required is to multiply two values and then divide them by 16. The multiplication is handled via a subroutine which is also shown in Figure 11. The division by 16 can be performed by a four place right shift followed by a rounding operation. The routine shown will handle a monotonic increasing function of a single independent variable. Fairly simple modifications are required for nonmonotonic functions. Functions of two variables can be handled by interpolating on a plane rather than along a straight line. Although this is more time consuming, requiring an interpolation for each of the independent variables and a third to interpolate the final answer, it still provides a simple means of quickly calculating the required function. The use of tables can offer a powerful technique for function evaluation to the designer.

**RECEIVING SERIAL CODE—BASIC APPROACHES**

Many microprocessor based systems require some form of serial communication. Serial communication is extensively used because it allows two or more pieces of equipment to exchange information with a minimal number of interconnecting wires. The minimization of interconnecting wires results in simpler, cheaper, interconnects because fewer (or smaller) cables and connectors are required. Since the required number of drivers and receivers required is reduced, it can become economically feasible to provide much higher noise immunity.
with more sophisticated (and expensive) line terminators. The final, and usually most persuasive, argument in favor of serial communication is that it may be the only method available to accomplish the job. The obvious example of this is telecommunications where it is necessary to encode parallel information into serial format in order to communicate via the telephone network. The intent of this section is to show how the facilities of the MCS-48™ can be brought to bear on the problem of serial communication.

Figure 12. Serial ASCII Code

Probably the most common form of serial communication is that used by the ubiquitous Teletype® serial ASCII. This format, shown in Figure 12, consists of a START bit (0 or SPACE) followed by eight data bits which are in turn followed by two STOP bits (1 or MARK). In actual practice the eighth data bit usually consists of even parity on the remaining seven data bits; for the purposes of this discussion the eighth bit will be considered only as data. A minor variation of this format deletes one of the STOP bits. An algorithm which might be used to sample serial data under software control using a microprocessor is shown in Figure 13. The basic intent of this algorithm is to minimize the effects of distortion and transmission rate variations on the reliability of the communication by sampling each data bit as close to its center as possible. Upon entry to this routine the software first samples the incoming data in a tight loop until it is sensed as a MARK (logical one). As soon as a MARK is detected, a second loop is entered during which the software waits until the received data goes to a SPACE (logical zero). The purpose of this construction is to detect as accurately as possible the leading edge of the START bit. This instant of time will be used as a reference point for sampling all of the following bits in the character. After sensing the leading edge of the START bit a wait of one half the expected bit time is implemented. The period of the incoming signal is called P for convenience. At the end of this wait the serial line is tested—if it is MARK then the START bit was
invalid and the process is reinitialized. If the line is still a SPACE, then the START bit is assumed to be valid and a delay of one bit time is started. At the completion of the delay the first data bit is sampled and a new delay of one bit time is initiated. This process is repeated until all eight data bits have been sampled. The last bit sampled is checked to determine if it is a valid STOP bit (a MARK). If it is, the character is assumed to be valid; if it is not, the character has a framing error and is probably invalid. A listing of a program which implements the above procedure is shown in Figure 14.

A disadvantage of the approach outlined in Figure 13 is that while the processor is inputting data serially it must totally dedicate itself to this task. Accurate timing can only be maintained if the program remains in a tight wait loop without allowing itself to be diverted to other functions. During reception of a character from a Teletype the processor will spend only a 100µsecs or so processing the data and the rest of the 100 milliseconds waiting to do the processing at the right time. This lack of efficiency (approximately 0.1%) in the utilization of processing power is why devices such as the 8251 USART find broad application in microprocessor systems.

The 8251 USART is simple to interface to the MSC-48. Figure 15 shows such an interface. The USART requires a high speed clock (CLK), an initialization signal (RESET), data clocks (TxC and RxC), and data in order to operate. A circuit showing the connection of an 8748 to an 8251 USART is shown in Figure 15. In the circuit shown the high speed clock (which is used for internal sequencing by the USART) is provided by con-

Figure 13. Sample Serial Input Routine

Figure 14. Simple Serial Input

All mnemonics copyrighted © Intel Corporation 1976.
The initialization signal to the USART (RESET) is provided programatically by manipulation of bit 5 of port 2. It was necessary to place the reset of the 8251 under program control for two reasons. The first reason is that the MCS-48 does not supply a reset signal to other devices. The reason for this is that it was felt to be more useful to provide another pin of I/O function instead of a RESET OUT signal from the MCS-48. Although this situation could have been circumvented by the use of an externally generated reset which drove both the MCS-48 and the 8251, the second reason for program control of the reset to the USART still stands. The USART requires the presence of the CLK signal during reset in order to properly initialize itself. The ENTO CLK instruction which the MCS-48 must execute before the 8251 will receive the CLK can obviously not be executed until after the system reset has ended. Reset of the USART can be accomplished by the following code segment:

```
ENTO CLK ; TURN ON CLOCK
ORL P2, #00100000B ; START RESET
MOV R2, #DELAY ; DELAY USART
LOOP: DJNZ R2, LOOP ; RESET TIME
ANL P2, #11011111B ; END RESET
```

This code first enables the clock, then asserts the reset signal of a time period determined by the constant DELAY. The delay invoked is (10 + 5*DELAY) microseconds for DELAY > 0. The USART requires a reset of approximately 6 CLK periods so DELAY is chosen to be 1 which ensures adequate reset timing. Note that for delays this short, NOP instructions could also be used to time the pulse.

The data clocks required by the USART are provided by the modem if the USART is operated in the synchronous mode. In the more common asynchronous mode, however, these clocks must be provided by circuitry associated with the 8251.

The 5.9904 MHz crystal was chosen because the resulting 1.9968 MHz clock to the USART can be evenly divided to provide transmit and receive clocks to the USART. Assuming the USART is in the x16 mode (i.e. it requires data clocks 16 times the baud rate) the 1.9968 MHz signal can be divided by 13 to generate the proper clock rate for 9600 baud operation. This 9600 baud clock can be further divided to give 4800, 2400, 1200, 600, and 300 baud signals. The 1200 baud signal can be divided by 11 to give a 109.1 baud signal which is within 1% of the 110 baud required by Teletypes.

The MCS-48 communicates with the 8251 in a memory mapped mode (i.e. as if the 8251 were external RAM). The instructions available to do this are MOVX @Rj, A which stores the contents of the accumulator at the external RAM location addressed by Rj (j=0 or 1), and its complement, the MOVX A, @Rj instruction which moves data from the external RAM into the accumulator. Since the MCS-48 multiplexes addresses and data on the same eight bit bus an external latch would be required in order to address the USART with
to add the circuitry necessary to use R0 or R1 to address the peripheral devices. The circuitry which has to be added to Figure 15 in order to make use of R0 or R1 to address the USART is shown in Figure 17. Note that only the changes to Figure 15 are shown. The additional component required is the 8212 eight bit latch. This latch is loaded, whenever a valid address is on the bus by the Address Latch Enable (ALE) signal provided by the MCS-48. During an external read or write cycle this address is used to address the 8251 in a linear select mode. In the circuit shown, the 8251 will be selected by any address with bit 1 a logical zero (XXXXXX0X) and the selection of control or data transfer (C/D) will be based on bit zero of the address obtained from R0 or R1. Figure 18 shows the program of Figure 16 modified to utilize the addressing inherent in the MOVX instructions.

R0 or R1. In order to minimize the circuitry in Figure 15 an approach utilizing some of the I/O pins of the MCS-48 to address the 8251 was chosen instead. By connecting the chip select (CS) input of the 8251 to bit 7 of port 2 (P27) and similarly connecting the C/D address line of the 8251 to bit 6 of port 2 (P26) it is possible to address the 8251 without using R0 or R1. The instruction sequence to access the 8251 is to first reset P27 and set P26 to the appropriate state, use a MOVX instruction to perform the appropriate operation, and then finally set P27 to deselect the 8251. As a concrete example of this addressing, Figure 16 shows the code necessary to initialize the 8251 and output an incrementing test pattern on a status driven basis. If more than one 8251 were to be added to the MCS-48, or if other types of peripheral circuitry would be required (e.g., an 8253 timer to generate the data clocks) it would probably become desirable

**Figure 16. 8251 Test Program**

```assembly
LOC OBJ SEQ SOURCE STATEMENT
1 ;-----------------------------------------------------
2 | 1:  | SERIAL TEST |
3 | 2:  | THIS CODE INITIALIZES THE USART |
4 | 3:  | AND TRANSMITS AN INCREMENTING |
5 | 4:  | PATTERN. HARDWARE SHOWN IF FIG 15. |
6 | 5:  | |
7 | 6:  | |
8 | 7:  | |
9 | 8:  | |
10 | 9:  | |

8209 11 MOVX A, #80H ; USART RESET ADDRESS
820A 12 DJNZ A, #80H ; USART RESET DELAY
820B 13 MOVX @R0, A ; USART CONTROL ADDRESS
820C 14 MOVX A, #80H ; USART MODE
820D 15 MOVX A, #80H ; USART MODE
820E 16 MOVX A, #80H ; USART MODE
820F 17 DJNZ A, #80H ; TESTING VALUE
8210 18 MOVX A, #80H ; CHANGES 320D TO DATA CHANNEL
8211 19 DJNZ A, #80H ;|
8212 20 DJNZ A, #80H ;|
8213 21 DJNZ A, #80H ;|
8214 22 DJNZ A, #80H ;|
8215 23 DJNZ A, #80H ;|
8216 24 DJNZ A, #80H ;|
8217 25 DJNZ A, #80H ;|
8218 26 DJNZ A, #80H ;|
8219 27 DJNZ A, #80H ;|
821A 28 DJNZ A, #80H ;|
821B 29 DJNZ A, #80H ;|
821C 30 DJNZ A, #80H ;|
821D 31 DJNZ A, #80H ;|
821E 32 DJNZ A, #80H ;|
821F 33 DJNZ A, #80H ;|
8220 34 DJNZ A, #80H ;|
8221 35 DJNZ A, #80H ;|
8222 36 DJNZ A, #80H ;|
8223 37 DJNZ A, #80H ;|
8224 38 DJNZ A, #80H ;|
8225 39 DJNZ A, #80H ;|
8226 40 DJNZ A, #80H ;|
8227 41 DJNZ A, #80H ;|
8228 42 DJNZ A, #80H ;|
8229 43 DJNZ A, #80H ;|
822A 44 DJNZ A, #80H ;|
822B 45 DJNZ A, #80H ;|
822C 46 DJNZ A, #80H ;|
822D 47 DJNZ A, #80H ;|
822E 48 DJNZ A, #80H ;|
822F 49 DJNZ A, #80H ;|
8230 50 DJNZ A, #80H ;|
8231 51 DJNZ A, #80H ;|
8232 52 DJNZ A, #80H ;|
8233 53 DJNZ A, #80H ;|
8234 54 DJNZ A, #80H ;|
8235 55 DJNZ A, #80H ;|
8236 56 DJNZ A, #80H ;|
8237 57 DJNZ A, #80H ;|
8238 58 DJNZ A, #80H ;|
8239 59 DJNZ A, #80H ;|
823A 60 DJNZ A, #80H ;|
823B 61 DJNZ A, #80H ;|
823C 62 DJNZ A, #80H ;|
823D 63 DJNZ A, #80H ;|
823E 64 DJNZ A, #80H ;|
823F 65 DJNZ A, #80H ;|
```

**Figure 17. Modified MCS-48 to 8251 Interface**

**RECEIVING SERIAL CODE—A MORE SOPHISTICATED ALGORITHM**

Although the USART does an admirable job of performing the serial I/O function for the MCS-48™, there are some situations where it can not be used. These situations may be caused by economic factors, such as an extremely cost sensitive design, or because the code which must be utilized cannot be accommodated by the USART. An example of such a code will be discussed later. Recall that the principal objection to the approach to serial input shown in Figure 13 was that it consumes much of the processor’s power by merely spinning in loops in order to wait preset time delays.
The timer resident on the MCS-48 provides a solution to this problem. Instead of spinning in a loop the program can set the timer for a given interval, start it, and proceed to other tasks. When the timer overflows, an interrupt will be generated to notify the software that the present time period has elapsed. An extension of the algorithm of Figure 13 which uses the timer in this fashion in shown in Figure 19. This algorithm is identical to the preceding one up until the detection of the leading edge of the start bit. At this point the timer is set to one half of the bit time (P) and a return is made to the calling program which can start additional processing. At the completion of this time interval a timer overflow interrupt is generated. When the first interrupt is detected, the serial line is checked to ensure that it is in a spacing condition (valid START bit). If it is, the timer is set to P (to sample the middle of the first data bit) and a return is made to the program which was running when the interrupt occurred. If the serial line has returned to the MARK state, a status flag is set to indicate an error and a return is made. On subsequent interrupt detection, the data is sampled, the timer is reinitiated, and control is returned to the program which was running when the interrupt occurred. When the last (i.e. STOP) bit is detected a completion flag is set and a return is made to the program running when the timer overflow occurred. By periodically checking the error and completion flags the running program can determine when the interrupt driven receive program has a character assembled for it.

**Figure 18. Modified 8251 Test Program**

Using the timer to implement time delays as shown in Figure 19 results in considerable savings in processing time; two problems remain, however, which must be solved before an adequate software solution to the problem of receiving serial code can be found. The first problem is that even though the delays between bit samples are implemented via the timer rather than program loops the loop construction is still used to detect the leading edge of
the START bit. Although this results in the waste of processing power, the second problem is even more serious. For longer messages the required accuracy of the clocks becomes more and more stringent. Using the sampling technique discussed a cumulative error of one half a bit time in the time at which a bit sample is taken will result in erroneous reception. The maximum timing error which can be tolerated and yet still allow proper detection of an 11 bit ASCII character is then:

\[ E_{\text{max}} = \frac{0.5 \times \text{BIT TIME}}{\text{CHARACTER TIME}} - \frac{0.5 \times \text{P}}{11 \times \text{P}} = 4.5\% \]

where \( P \) is the period of single bit. The corresponding calculation for a 32 bit character yields:

\[ E_{\text{max}} = \frac{0.5 \times \text{P}}{32 \times \text{P}} = 1.6\% \]

Since this calculation does not allow for distortion on the signals, it is obvious that either extremely stable clocks will be required or a more tolerant algorithm must be devised. This problem is particularly serious at relatively high baud rates where the resolution of the counter (80usecs with a 6 MHz crystal) becomes a significant percentage of the period of the received signal. At the 110 baud rate of the Teletype the 80usec resolution of the clock allows a maximum accuracy of 0.33%; at 2400 baud this figure is reduced to 3.8%.

Both efficient detection of the start bit and increased timing accuracy can be obtained if the MCS-48 can detect edges on the incoming received data (RxD). A hardware construct which allows this is shown in Figure 20.

The received data (RxD) is Exclusive NORed with bit seven of port two and fed into the TEST (TI) pin of the MCS-48. By manipulating P27 the program can now cause TI to be either Rx\(D \) or R\(xD \). (If P27 = 1 then TI = R\(xD \); if P27 = 0 then TI = RxD.) Note that not only can TI be tested directly by the software but that it is the input which is used when the MCS-48 timer is in the event counter mode. The significance of this will be discussed later. The relationship between TI, P27, and RxD is given by the Boolean expression:

\[ T_I = P_{27} \cdot RxD + P_{27} \cdot \overline{RxD} \]

Figure 21 flowcharts a means of utilizing this hardware construct to avoid the necessity of wasting time in program loops to detect the leading edge of the start bit. The receive operation is initialized when the program desiring to receive serial data calls the INIT subroutine (Figure 21a). Since INIT is going to manipulate the timer the first action it performs is to disable the timer overflow interrupt. Its next step is to set P27 to a logical 1. Setting P27 in this manner causes the TEST 1 input to the MCS-48 to follow RxD. By setting up the receive circuitry in this manner a high to low transition will occur on TEST 1 when the RxD goes from the MARKING to SPACING state (i.e. the START
bit occurs). By setting the timer to $0FFH$ and enabling it in the event count mode, the INIT routine sets up the MCS-48 to generate a timer overflow interrupt on the next MARK to SPACE transition of RxD (the TEST 1 input doubles as the event counter input). Before returning to the calling program the INIT routine sets a flag (RDF) which will be cleared by the receive program when the requested receive operation is complete. INIT also sets a value into a register called BCOUNT. The receive program interprets BCOUNT as follows:

<table>
<thead>
<tr>
<th>BCOUNT</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bits remaining to receive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If set indicates that the START bit has not yet been detected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If set indicates that the START bit has not yet been verified</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In order to request the reception of the 11 bit ASCII code INIT would set BCOUNT to $1100101B$. The start bit has been neither verified nor detected and 11 bits ($1011B$) are required.

After INIT is called the reception of the individual serial data bits will proceed on an interrupt driven basis until a complete character has been assembled. When this occurs the interrupt driven program will set the RDF (Receive Done Flag) to a zero to indicate that it has completed the requested operation and then terminate itself. The procedure which is used to accomplish this is shown in Figures 21b and 21c.

Since all operations of this program are the result of the occurrence of a timer overflow interrupt, it is necessary to briefly review the interrupt structure of the MCS-48. There are two sources of interrupt; an external interrupt which is the result of a logical zero signal applied to the INT pin of the MCS-48, and an internal interrupt which is caused by a timer overflow condition. The timer overflow occurs whenever the timer is incremented from $0FFH$ to zero whether it be in the timer or event count mode. When one of these events occurs the hardware in the MCS-48 forces the execution of a CALL. This CALL has a preset address of location 3 if it is due to the external interrupt and location 7 if it is due to a timer overflow. If both of these
events occur simultaneously the external interrupt will take precedence. The CALL automatically saves the contents of the program counter for the running program and its PSW (program status word) on a stack the hardware maintains in RAM locations 8-23. Although the hardware saves the program counter and PSW, it remains the responsibility of any interrupt driven software to make absolutely certain that it does not modify any memory locations or registers which are being used by the main program. The most convenient way of ensuring this in the MCS-48 is to dedicate the second bank of registers (RB1) to the interrupt driven program. One of these registers has to be used to save the accumulator (which is not part of the register bank) but seven registers remain; including two which can be used as pointers to the rest of the RAM (R0 and R1). Note that if this approach is taken then these registers have to be allocated between the program which services the external interrupt and the one which services the timer overflow. This problem is somewhat alleviated by a hardware lockout which prevents the timer overflow interrupt from interrupting the external interrupt service routine and vice versa. This is implemented by locking out new interrupts between the time an interrupt is recognized and the time a RETR instruction is executed. The RETR instruction is like a normal RET (return from subroutine) except that the PSW as well as the program counter is restored. The RETR instruction can be very much thought of as a return from interrupt instruction in the MCS-48.

The receive program under discussion uses register bank 1 in the manner described. Whenever a timer overflow occurs (e.g. on the next MARK to SPACE transition of RxD after INIT is called), control is passed (by the hardware generated CALL) to the point labeled TIMER OFLO in Figure 21b. This program segment immediately selects register bank 1 (RB1) and then saves the accumulator (A) in a location called ATEMP which is actually R7 of RB1. The program then tests bit seven of BCOUNT (R6 of RB1) to find out if a START bit has been verified (i.e. the edge of the START bit has first been detected and then verified to still be a SPACE one-half a bit time later. If BCOUNT [7] is a zero the START has been verified and the program proceeds to set the timer to P (the period of the serial bit), get the current serial data into the carry bit, and then shift the carry bit into a buffer. After saving the data the program decrements BCOUNT and tests it for zero. If BCOUNT is zero the receive operation is complete so the program sets RDF to a zero and disables timer overflow interrupts. Whether or not BCOUNT is zero, control is passed to EXIT where A is loaded with ATEMP and a RETR is executed. Note that since the state of the flip flop which selects RB1 is saved as part of the PSW, the execution of RETR automatically selects the register bank which was active when the interrupt occurred.

If BCOUNT [7] is still set when it is tested, control is passed to START (Figure 21c) where bit 6 is tested to determine if the START has been detected yet. If BCOUNT [6] is set it indicates that this is the first occurrence of a timer overflow since the receive process was initialized by the INIT subroutine. If this is so, the program assumes that the START bit has just started and therefore it sets the timer to one-half of a bit time (1/2 P), starts the timer in the timer mode, and clears BCOUNT [6] to indicate that the START bit has been detected. The next overflow will again result in the execution of the program in Figure 21b and again BCOUNT [7] will be found to be set. This time, however, BCOUNT [6] will be reset and the program will know that it should test the START bit to ensure that it is still a SPACE. This test is performed and if successful the timer is set for a bit period P and BCOUNT [7] is reset so that on the next occurrence of a timer overflow the program will know that it should start assembling serial bits into a character. If the test is unsuccessful, the subroutine INIT is used to reinitialize the receive program. In either case control is passed to EXIT where a return from interrupt mode occurs.

This receive program, listings of which appear in Figure 22, allows the reception of serial characters transparently to the main running software. After INIT is called the main program has only to check RDF periodically to find out if there is data in the buffer for it. It would be fairly easy to 'double buffer' this operation by providing a buffer which the receive program uses to deserialize the incoming code and a second buffer to store the assembled character. If the program would reinitialize itself upon completion, the reception of a string of characters could proceed in much the same way as it would if a status driven USART were being used.

Although this program solves the first problem of software controlled reception (lack of efficiency) the second problem—sensitivity to frequency variations—remains. An example of a code which would be susceptible to this problem is the 31,26 BCH code commonly used in supervisory control systems. (A supervisory control system is, in essence, a remote control system which allows a human or computer operator the control of a system via a serial communications link.) The BCH codes are used because of their error detection capabilities and are a class of cyclical redundancy
nominal BCH codes can detect all errors consisting of 2t error bits and all burst errors of mt or fewer bits. The 31,26 BCH code will therefore detect any erroneous messages with 1 or 2 errors or bursts of errors of less than 5 bits. The 31,26 format (shown in Figure 23) requires the reception of a start bit followed by 31 information bits, clearly beyond the capability of the USART but perhaps within reach of a program controlled approach using the MCS-48 itself.

All mnemonics copyrighted © Intel Corporation 1976.

---

Figure 22. Interrupt Driven Serial Receive Program
A concept which reduces sensitivity to frequency deviations and thus allows the reception of longer codes is shown pictorially in Figure 24. The first line of this timing chart shows an alternative ones and zeros pattern on the RxD with a period of 5 milliseconds. The second line shows that by sampling at a period of exactly 5 milliseconds the data can be properly interpreted. The third and fourth lines show the effects of sampling with a period of six and four milliseconds respectively. In either case, an error occurs at the third sample where both periods result in sampling on an edge of the RxD signal. The third line of Figure 24 shows a hybrid sampling scheme which, based on some additional information, switches sampling periods between the two values. As can be seen in Figure 24, the data is sampled with a 4 millisecond period until the sampling begins to fall behind the data; at this point the sampling period is increased to six milliseconds and the sampling first catches up and then passes the center point of the data. As soon as this happens, the sampling period reverts to the 4 millisecond period and the cycle repeats. It can be seen that this scheme sets up a pattern which repeats indefinitely and the data can be successfully sampled. Note that the sampling pattern established is alternating periods of four and six milliseconds. The average period of this pattern, as might be expected, is 5.5 m.sec. Line 5 of Figure 24 shows the effect of a change in transmission speed to a period of 5.5 m.sec with no change in the sampling time. The sampling is again successful but the new sampling pattern is 4-6-6-6; 4-6-6-6, etc. Note that the average sample is again equal to the period of the received data (5.5). While this scheme does seem to work, the question of what additional information is needed remains.

The MSC-48 must somehow decide when it is drifting out of synchronization and take corrective action. By referring back to Figure 24 it can be seen that if the MCS-48 could determine where the edges of RxD occurred with respect to its sampling times then the additional information would be available. As can be seen in the figure the choice of sampling period can be based on the following rule:

*If an edge on the RxD line occurs during the first half of the current sampling period, then use the short period for the next sample. If an edge occurs during the second half of the period, then use the long sampling period for the next sample.*

If the data on the RxD line does not change, of course, the MCS-48 will drift out of synchronization just as the original algorithm did. As long as edges occur on TxD, however, synchronization can be maintained. To maximize the allowable time between edges, the following addition could be made to the above rule:

*If no edge occurs on the RxD line during a sample, then change sampling period from short to long or vice versa.*

Note that this addition to the rule will result in using an average of the two sampling periods when no edge occurs for several bit times.

The edges of RxD can be easily detected by the use of the same structure (the Exclusive – NOR gate) which was added to the MCS-48 in Figure 20. This gate, which is used to detect the edge on RxD which begins the START bit, can naturally be used to detect any edge. Since the timer is being used to time the bit period, however, the event count input (T1) is not useful during the receive itself. By connecting the output of this gate, however, to the INT input to the MCS-48 (see Figure 25) it is possible to detect edges on RxD with the event counter when the program is trying to detect the START bit and by the external interrupt when the program is using the timer to control the sampling times.
A modification to the program of Figure 21 which implements this new sampling algorithm is shown in Figure 26. The first deviation from the original program is the addition of a routine (XISR, Figure 26a) which is called when an external interrupt occurs (i.e., when an edge occurs on RxD). This routine saves the status of the running program and then stores the current value of the timer register in a location called SNAP (R5 of RB1). After doing these operations the program complements bit 7 of port 2. Manipulating P27 in this manner will cause the Exclusive NOR gate to turn off the external interrupt and will set it up to generate another interrupt when the RxD line changes again (has another edge).

Because of this edge detection it is important to condition RxD with hardware filters to ensure that the edges of RxD are clean. Any ringing will cause repeated Calls to XISR and probable erroneous operation. The changes to the START process (Figure 26c) are two-fold; first the TIMER is set to one half the average of the two sample periods when the START bit is first detected (BCOUNT[6] = 1), and second the processing of the edge information is initialized by presetting SNAP and clearing P27.

SNAP is preset so that when the reception of data actually begins (Figure 26b BCOUNT[7] = 0), the decision block which tests SNAP against LIMIT will be initialized. This block actually compares the value in SNAP with a LIMIT value which is used to determine if the sampling point is ahead or behind the actual midpoint of the serial data. If the sampling is ahead then the timer is set for TMIN; if the sampling is behind then the timer is set for...
TMAX. By presetting SNAP in the manner shown in the flowcharts the second rule of the algorithm, (if no edge appears on the RxD line during a sample, then change the sampling periods short to long or vice versa) is automatically met. If an edge occurs then XISR will modify SNAP, if XISR is not invoked between two samples then the choice of timer periods will alternate. The only other significant change to the algorithm is that the INIT routine must now lock out all interrupts, not just the timer overflow interrupt, while it is operating. A program which uses this algorithm to receive a 32 bit message is shown in Figure 27.

Figure 27. Hybrid Sampling Program
Figure 27. Hybrid Sampling Program

All mnemonics copyrighted © Intel Corporation 1976.

1-22
TRANSMITTING SERIAL CODE

Serial transmission is conceptually far simpler than serial reception since no synchronization is required. All that is required is to use the timer to generate interrupts at the bit rate and present the character to be transmitted serially at an I/O pin. A program which does this is shown in Figure 28. The transmission of serial data becomes much more complicated if it must occur simultaneously with reception.

If both reception and transmission are to occur simultaneously then obviously contention will exist for the use of the timer.

TRANSMITTING SERIAL DATA

To be transmitted serially at a baud rate using the timer as a general clock which controls software maintained timers. The attainable baud rates using such techniques are, however, limited and the use of a 8251 USART is probably indicated in all but the most cost sensitive applications. An exception to this rule occurs when the system, although full duplex in nature, actually transmits the same data as it receives. An example of this is a microprocessor driving a terminal such as a Teletype. Although the circuit to the terminal is full duplex, the data that is transmitted is generally the same as that received. A minor modification to the program shown in Figure 26 would implement this mode of operation. The modification would be to the XISR routine and it would add the code necessary to place the TxD I/O pin in the same state as the RxD line. Since any change in RxD results in a call to XISR, this modification would cause the retransmission of any received data. Whenever it becomes necessary to transmit data which is not being received, the program of Figure 28 could be used in a half duplex manner.

```
LOC OBJ  SEQ  SOURCE STATEMENT

1      EA   ; SERIAL TRANSMIT ON THE PCON
2      EA   ; TO USE PUT A CHAIN IN Buf AND
3      EA   ; CLEAR R/D BFF, WHEN THE
4      EA   ; TRANSMIT IS READY FOR ANOTHER
5      EA   ; CHAIN IT WILL CLEAR CHAIN
6      EA   ; TRANSMIT IS DOUBLE BUFFERED.
7      EA   ;
8      EA   ;
9      EA
10     EA   18 ;
11     EA   12 ;
12     EA   13 ;
13     EA   14 ATEMP EQU R7   ; STORAGE FOR A DURING INT.
14     EA   15 PTOS EQU R6   ; PARALLEL TO SERIAL CONVERTER
15     EA   16 BUFT EQU R5   ; CHARACTER BUFFER
16     EA   17 CHARMV EQU R4   ; CHARACTER AVAILABLE FLAG
17     EA   18 COUNT EQU R3   ; BIT COUNTER
18     EA   19 CBIT EQU R2    ; MASK TO CLEAR T/R BFF IN P2
19     EA   20 SBIT EQU R1    ; MASK TO SET T/R BFF IN P2
20     EA   21 P EQU +41 ; PERIOD OF T/R
21     EA   22 ;
22     EA   23 ;
23     EA   24 ; CONTROL PASS MORE ON TIMER OVERFLOW
24     EA   25 ;
25     EA   26 ;
26     EA   27 ;
27     EA   28 ;
28     EA   29 ATEMP, A
29     EA   30 TPDLD: SEL. RB1 ; ENTER INTERRUPT MODE
30     EA   31 MOV ATEMP, A
31     EA   32 MOV A, P
32     EA   33 MOV A, P
33     EA   34 MOV A, P
34     EA   35 MOV A, P
35     EA   36 MOV A, P
36     EA
```

```
LOC OBJ  SEQ  SOURCE STATEMENT

#88T BA  37 IN A, P2
#88D D3H  38 MOV A, #0
#88B 3AH  39 MOV A, P2
#88D 3E0  40 OUT P2, A
#88B 3E5  41 JC BITON
#88B 3E6  42 JMP EXIT
#88B 3E7  43 BITON: DBL P2, #0BIT
#88B 3E8  44 EXIT: MOV A, ATEMP
#88B 3E9  45 RET
#88B 3EB  46 ;
#88B 3EC  47 ;
#88B 3ED  48 ; BIT ROUTINE
#88B 3EE  49 ; P/X T HE NEXT BIT TO TRANSMIT
#88B 3EF  50 ;
#88B 3F0  51 ;
#88B 3F1  52 BIT: MOV A, COUNT
#88B 3F2  53 MOV A, #0
#88B 3F3  54 MOV A, PTOS
#88B 3F4  55 MOV A, R2
#88B 3F5  56 MOV A, #0
#88B 3F6  57 MOV A, P2
#88B 3F7  58 MOV A, #0
#88B 3F8  59 MOV A, #0
#88B 3F9  60 MOV A, #0
#88B 3FA  61 MOV A, #0
#88B 3FB  62 MOV A, #0
#88B 3FC  63 MOV A, #0
#88B 3FD  64 MOV A, #0
#88B 3FE  65 MOV A, #0
#88B 3FF  66 MOV A, #0
#88B 400  67 MOV A, #0
#88B 401  68 MOV A, #0
#88B 402  69 MOV A, #0
#88B 403  70 MOV A, #0
#88B 404  71 MOV A, #0
#88B 405  72 MOV A, #0
#88B 406  73 MOV A, #0
```

Figure 28. Serial Transmission

All mnemonics copyrighted © Intel Corporation 1976.
GENERATING PARITY

Many communications schemes require the generation and checking of parity. If a USART is used it can be programmed to automatically generate and check parity. If the communications is handled by software within the MCS-48™ then the program must perform parity calculations. Calculating parity is easy if one remembers what parity really means. A character has even parity if the number of one bits in it is even. A character has odd parity if it has an odd number of ones. The program segment shown in Figure 29 can be caused to calculate parity. It starts by setting a loop count to eight and clearing the CARRY flag. After this initialization a loop is executed eight times. During each execution the accumulator is rotated and the least significant bit is tested. If the bit is a zero the CARRY flag is complemented, if the bit is a one no further action is taken. Since an even number of zeros implies an even number of ones for an eight bit character, after all eight loops have been accomplished the CARRY bit will be set if an odd number of ones were encountered; it will be reset if the number were even. Since the RR instruction does not involve CARRY the net result of executing this program loop is to set CARRY if parity is odd without effecting the character in the accumulator.

CONCLUSION

This Application Note has presented a very small sampling of the application techniques possible with the MCS-48™ family. The application of this new single chip computer system to tasks which have not yet yielded to the power of the microprocessor will present a fascinating challenge to the system designer.
Keyboard/Display Scanning
With Intel's MCS-48™
Microcomputers

John Wharton
Microcomputer Applications
INTRODUCTION

This application note presents a software package for interfacing members of Intel's MCS-48™ family of single-chip microcomputers with keyboards and displays using a minimum of external components. Because of the similarity of the architectures of the various members of the family (the 8035, 8048, 8748, 8039, 8049, 8021, and 8022 microcomputers; also the 8041 and 8741 universal peripheral interfaces in the UPI-41® family), the code included here could run with minor modifications on any member of the family.

Since keyboard and display logic can be just one of several functions handled by a microprocessor, the added cost of including these functions in a system is minimal. In fact, considering the extremely low cost of standard X-Y matrix keyboards and integrated displays, their use is often more cost effective than even a handful of discrete switches and indicators. Thus, the additional flexibility of keyboard input and display output can be added to inexpensive consumer products (such as games, clocks, thermostats, tape recorders, etc.), while producing a net savings in system cost.

Since each potential application will have its own unique combination of keys and display characters, the program is written so that very little modification is needed to interface it with a wide variety of hardware configurations. In general, the only changes required are within the set of initial EQUates at the beginning of the program.

Along with the basic software for driving a multiplexed display and/or scanning and debouncing an X-Y matrix of key switches, a collection of utility subroutines is also included for implementing the most commonly used keyboard and display utility functions, such as copying simple messages onto the display or determining the encoded value of each key in the key matrix. As a result of the versatile architecture and applications-oriented instruction set of the MCS-48 family, the entire package fits into about 250 bytes of internal program ROM or EPROM, leaving the rest of the ROM space for the program to cook the perfect piece of toast, or whatever. By tailoring the software to match a known hardware configuration, or by selecting only those functions needed for a given application, the program size could be even further reduced.

Since what is being presented in this application note is a software package, rather than the usual hardware/software system design, the format of this note is somewhat different from most — it consists primarily of a long program listing reproduced in the following pages. For the most part, the listing is self-explanatory, with comments introducing each subroutine and major code segment. Some parts of this introduction are reproduced in the program listing itself, explaining the configuration of the prototype system. However, an additional bit of explanation would make the listing easier to understand, especially for those readers unfamiliar with the concept of multiplexed displays and keyboards.

In traditional digital system design, various hardware registers or counters were used to hold binary or BCD values which had to be conveyed to the user. The standard way of presenting this information was by connecting each register to a seven-segment encoder (such as the 7447) driving a single display character, as represented by Figure 1. Thus, two ICs, seven current-limiting resistors, and about 45 solder joints were required for each digit of output. Consider how traditional techniques might be applied in designing a microprocessor system: the designer could add a latch, encoder, and resistors for each digit of the display. Still another latch and decoder could be used to turn on one of the decimal points (if used). The characters displayed could only be a sequence of decimal digits. In the same vein, a large matrix of key switches could be read by installing an MSI TTL priority encoder read by an additional input port. Not only would all this use a lot of extra I/O ports and increase the system price and part count drastically, but the flexibility and reliability of the system would be greatly reduced.

In Figure 1, the wrong way of designing multiple-digit displays is shown. Instead of providing a separate latch and decoder for each digit of output, the entire seven-segment display is driven by a single decoder/driver chip. This is much more economical, but it requires that the designer choose a display with a pin-out compatible with the decoder/driver, and also that the display be used in a way which allows all bits of each character to be turned on simultaneously. This is usually not the case.
Instead, a scheme of time-multiplexing the display can be used to decrease costs, part count, and interconnections, while allowing a wider range of character types to be used on the display. The techniques used here are fairly typical of today's integrated subsystems designed especially for controlling keyboards and displays (such as in calculators or the Intel® 4269, 8278, and 8279 Keyboard/Display Controller Devices).

In a multiplexed display, all the segments of all the characters are interconnected in a regular two-dimensional array. One terminal of each segment is in common with the other segments of the same character; the other terminal is connected with the same segments of the other characters. This is represented schematically in Figure 2. A digit driver or segment driver is needed for each of these common lines.

![Figure 2. Schematic Representation of 6-Digit, 7-Segment Common-Cathod LED Multiplexed Display](image)

The various characters of the display are not all on at once; rather, only one character at a time is energized. As each character is enabled, some combination of segment drivers is turned on, with the result that a digit appears on the enabled character. (For example, in Figure 3, if segment drivers 'a', 'b', and 'c' were on when character position #6 was enabled, the digit '7' would appear in the left-most place.) Each character is enabled in this way, in sequence, at a rate fast enough to ensure that the display characters seem to be on constantly, with no appearance of flashing or flickering.

In the system presented here, these rapid modifications to the display are all made under the control of the MCS-48™ microcomputer. At periodic intervals the computer quickly turns off all display segments, disables the character now being displayed and enables the next, looks up the pattern of segments for the next character to be displayed, and turns on the appropriate segments. With the next character now turned on, the processor may now resume whatever it had been doing before. The whole display updating task consumes only a small fraction of the processor's time.

![Figure 3. Segment and Digit Drivers used with 6-Position, 7-Segment LED Display](image)

Moreover, since the computer rather than a standard decoder circuit is used to turn the segments off and on, patterns for characters other than decimal digits may be included in the display. Hexadecimal characters, special symbols, and many letters of the alphabet are possible. With sufficient imagination this feature can be exploited for some applications, as suggested by the examples in Figure 4.

![Figure 4. Examples of Typical Messages Possible with Simple 7-Segment Displays](image)
As each character of the display is turned on, the same signal may be used to enable one row of the key matrix. Any keys in that row which are being pressed at the time will then pass the signal on to one of several "return lines", one corresponding to each column of the matrix. (See Figure 5.) By reading the state of these control lines, and knowing which row is enabled, it is possible to compute which (if any) of the keys are down. Note that the keys need not be physically arranged in a rectangular array; Figure 5 is merely a schematic.

Since each character is on for only a small fraction of the total display cycle, its segments must be driven with a proportionately higher current so that their brightness averages out over time. This requires character and segment drivers which can handle higher than normal levels of current. Various types of drivers can be used, ranging from specially designed circuits to integrated or discrete transistor arrays. The selection depends on several factors, including the type of display being used (LED, vacuum fluorescent, neon, etc.), its size, the number of characters, and the polarity of the individual segments. Some drivers have active high inputs, some active low. Some invert their input logic levels, some do not. Some require insignificant input currents, some present a considerable load. Some systems use external logic to enable one of N characters or to produce the appropriate segment pattern for a given digit, some systems implement these functions through software.

Because of these and the other variables which make each application unique, provisions are made in the first page of symbol EQUates to allow the user to specify such things as the number of characters in the display or the polarity of the drivers used, and the program will be assembled accordingly. The display is refreshed on each timer interrupt, which occurs every 32 x (TICK) machine cycles. One machine cycle occurs every 30 crystal oscillations for the 8021 and 8022, or every 15 oscillations for all other members of the family.) A more detailed explanation of these variables is included in the listing.

Port assignment is also at the discretion of the user — all port references in the listing are "logical" rather than physical port names. The port used to specify which character is enabled is referred to as "PDPULIT". The output segment pattern is written to "PSGMNT" and the keyboard return lines are read by "PINPUT". These logical port names may be assigned to whichever ports the user pleases.

By way of example, the breadboard used to develop and debug this software used a matrix of 16 single-pole pushbuttons and an 8-character common-cathode LED display with right-hand decimal point. No decoders external to the 8748 microcomputer were used; all logic was handled through software. PDIGIT was the 8-bit bus, PSGMNT was port 1, and PINPUT was port 2. The drivers used were 75491 and 75492 logically non-inverting buffers; high level inputs were used to turn a segment or character on. Pull-up resistors were used on the 8748 output lines to source the current levels needed by the buffers. The 8748 was socketed on the breadboard, and was driven with an inexpensive 3.59 MHz television crystal. The short test program included in this listing was used to echo key depressions as they were detected, and to invoke four demonstration subroutines. A summary of the subroutines included in this listing with a short explanation of the function of each is included in Figure 6; Figure 7 shows how the various utilities interact.

**KBDIN** Keyboard Input. Waits until one keystroke input has been received from the keyboard; determines the meaning or legend of that key and returns with the encoded value in the accumulator.

**CLEAR** Blank out the display.

**ENCACC** Encode accumulator with bit pattern corresponding to the segment pattern needed by the display to represent that symbol or character. Uses the value of the accumulator when called to access a table containing the patterns for all legal input values.

**WDISP** Write into Display. Writes the bit pattern in the accumulator into the next character position of the display. Maintains a character position counter so that repeated calls will automatically write characters into sequential positions.

**RENTY** Right-hand Entry. Stores the accumulator segment pattern in the display in the right-most character position. Shifts all other characters to the left one place.

**PRINT** Print a string of arbitrary characters onto the display. Useful for prompting messages, warnings, etc. Uses a table of segment patterns in ROM, so that messages will not be restricted to numbers, letters, etc.

**FILL** Fill the display with the character pattern in the accumulator. Useful for writing dashes, segment test patterns, etc., into all character positions.

**ECHO** Wait for a key to be pressed by the operator and write that key onto the display. Used for providing feedback to the operator when entering numeric data, etc.

**RDPADD** Adds or deletes a decimal point to the character at the right-hand side of the display, for entering floating point numbers.

**HOLD** Called when a key is known to be down. Does not return until all keys have been released. Used for organ-type keyboards, or when some action should not be initiated until the key invoking that action has been released.

**DELAY** Provides a crude real-time delay corresponding to the value of the accumulator. Can be used to cause display characters to blink, to momentarily flash information, to enable a buzzer, etc. Could also be used by the program when delays are needed, such as to slow down the computer reaction rate while playing a game against the human operator.

*Figure 5. Schematic of X-Y Matrix Multiplexed Keyboard*

*Figure 6. Utility Subroutine Definitions*
Figure 7. Subroutine Interrelationships

Figure 8 Prototype System Schematic
LOC 081 560  SOURCE STATEMENT

1  #MACROFILE NPREF
2  TITLE('AP40 INTEL MCS-48 KEYBOARD/DISPLAY APPLICATION NOTE APPENDIX')

4  THE FOLLOWING SOFTWARE PACKAGE PROVIDES A SEVEN SEGMENT DISPLAY
5  INTERFACE FOR MICROCOMPUTERS IN THE INTEL MCS-48 FAMILY
6  THE CODE IS WRITTEN SUCH THAT VARIOUS HARDWARE
7  CONFIGURATIONS CAN BE ACCOMMODATED BY REDEFINING THE INITIAL VARIABLES
8  IN MOST SITUATIONS, THE KEYBOARD/DISPLAY INTERFACE WILL BE REQUIRED TO
9  IMPLEMENT MORE SOPHISTICATED SINGLE-CHIP SYSTEMS (CALCULATORS, SCALES, CLIKS,
10  ETC.). WITH SECTIONS OF THE FOLLOWING CODE SELECTED AND MODIFIED AS NECESSARY.
11  FOR EACH APPLICATION.
12  
13  1. A SINGLE SUBROUTINE (CALLED REFRESH) IS USED TO IMPLEMENT BOTH THE DISPLAY
14  MULTIPLEXING AND KEYBOARD SCANNING, USING THE SAME SIGNAL BOTH TO ENABLE
15  ONE CHARACTER OF THE DISPLAY AND TO STROBE ONE ROW OF THE X-Y KEY MATRICES.
16  THE SUBROUTINE MUST BE CALLED SUFFICIENTLY OFTEN TO ENSURE THE DISPLAY
17  CHARACTERS DO NOT FLICKER- AT LEAST 50 COMPLETE DISPLAY SCANS PER SECOND
18  TO ACCOMMODATE SWITCHES OF ARBITRARY CHEAPNESS, THE DEBOUNCE TIME CAN BE
19  SET TO BE ANY DESIRED NUMBER OF COMPLETE SCANS.
20  THUS THE DEBOUNCE TIME IS A FUNCTION OF BOTH THE SCAN RATE AND THE VALUE
21  OF CONSTANT 'DEBOUNCE'.
22  
23  IN THIS LISTING, THE INTERNAL TIMER IS USED TO GENERATE INTERRUPTS THAT
24  SERVE AS A TIME BASE FOR THE REFRESH SUBROUTINE.
25  ALTERNATE TIME BASES MIGHT BE AN EXTERNAL OSCILLATOR (DRIVING THE INTERRUPT
26  PIN OR POLLED BY A TEST OR INPUT PIN), A SOFTWARE DELAY LOOP IN THE BACKGROUND
27  PROGRAM, OR PERIODIC CALLS TO THE SUBROUTINE FROM THROUGHOUT THE USER'S PROGRAM
28  AT APPROPRIATE PLACES.
29  IN THESE CASES, THE CODE STARTING AT LABEL TTINT (TIMER INTERRUPT) AND TIPRT.
30  (TIPRT RETURN) COULD STILL BE USED TO SAVE AND RESTORE ACCUMULATOR CONTENTS
31  THE INTERRUPT SERVICING ROUTINE SELECTS REGISTER BANK 1
32  FOR THE NEEDED REGISTERS.
33  
34  
35  WRITTEN BY JOHN WHARTON. INTEL SINGLE-CHIP COMPUTER APPLICATIONS
36  
37  SELECT
LOC OBJ SED SOURCE STATEMENT

38. IN THIS IMPLEMENTATION OF THE DISPLAY SCAN, IT IS ASSUMED THAT THERE WILL
39. BE RELATIVELY LITTLE I/O OTHER THAN FOR THE KEYBOARD/DISP.
40. IF THIS IS THE CASE, THEN THERE IS NO NEED FOR ANY ADDITIONAL EXTERNAL
41. LOGIC (SUCH AS ONE-OR-EIGHT DECODERS OR SEVEN-SEGMENT ENCODERS), THOUGH
42. THERE WILL STILL BE A NEED FOR CURRENT OR VOLTAGE DRIVERS. ACCORDING TO
43. THE TYPE OF DISPLAY BEING USED
44. .
45. IN THIS LISTING, THE PROCESSOR I/O PORTS ARE LOGICALLY DIVIDED AS FOLLOWS
46. .
47. -DIGIT-EIGHT BIT PORT USED TO ENABLE, ONE AT A TIME, THE INDIVIDUAL
48. . CHARACTERS OF AN EIGHT DIGIT SEVEN-SEGMENT DISPLAY, WHILE ALSO
49. . STROBING THE ROWS OF AN X-Y MATRIX KEYBOARD.
50. . BIT7 ENABLES THE LEFTMOST CHARACTER AND THE BOTTOM ROW OF THE 16;
51. . BIT4 ENABLES THE TOP ROW OF THE 4X4 KEI AND THE FOURTH CHARACTER;
52. . BIT0 ENABLES THE RIGHTMOST CHARACTER
53. . (A 4X8 KEYBOARD COULD BE STROBED BY ALSO USING BIT3-BIT0
54. . AND EXTENDING OR ELIMINATING THE TABLE: "LEGEND")
55. . THE ENABLING OF ONE BIT (ACTIVE HIGH OR LOW) IS ACCOMODATED BY
56. . ACCESSING A LOOK-UP TABLE CALLED CHRSB.
57. . THIS TECHNIQUE TAKES ABOUT FOUR TIMES MORE ROM THAN A TECHNIQUE
58. . OF ROTATING A 'ONE' THROUGH A FIELD OF 'CEROES' IN THE ACC.
59. . AN APPROPRIATE NUMBER OF TIMES; BUT IT ALLOWS SOME ADDITIONAL
60. . FLEXIBILITY IF THE DRIVERS BEING USED HAVE A COMBINATORIAL INPUT
62. . THE CHRSB TABLE COULD PROVIDE ENCODED OUTPUTS. NINE DIGITS, FOR
63. . EXAMPLE, COULD BE ENABLED WITH SIX BITS OR (BUFFERED) OUTPUT
64. . (.01000.00101.00010.01000.01001.01010.10001.10001.10000)
65. . IF I/O LINES NEED TO BE CONSERVED, OR IF MANY DIGITS
66. . MUST BE DISPLAYED, AN EXTERNAL DECODER COULD BE ADDED TO THE SYSTEM
67. . DURING CHARACTER TRANSITIONS A 'BLANK' CHARACTER IS
68. . EXPLICITLY WRITTEN TO THE DISPLAY. THEREFORE
69. . THERE WILL BE NO CHARACTER 'SHADOWING' CAUSED BY THE
70. . FACT THAT THE HARDWARE OR SOFTWARE DECODER KEEPS ONE
71. . OUTPUT, AND THUS ONE CHARACTER, ACTIVE AT ALL TIMES
72. .
73. -DIGIT-EIGHT BIT PORT TO ENABLE THE SEVEN SEGMENTS & DP OF A STANDARD
74. . DISPLAY.
75. . BIT7-BIT0 CORRESPOND TO THE DP AND SEGMENTS 0 THROUGH 9 RESPECTIVELY.
76. . IT IS POSSIBLE TO ACCOMODATE
77. . DRIVERS WHICH ARE EITHER LOGICALLY INVERTING OR NON-INVERTING BY
78. . SETTING VARIABLE 'SEGDP' (SEGMENT POLARITY)
79. . NOTE THAT BY HAVING ARBITRARY CONTROL OVER EACH SEGMENT, NON-NUMERIC
80. . CHARACTERS CAN BE REPRESENTED ON A SEVEN SEGMENT DISPLAY.
81. . AS SHOWN IN EXAMPLE SUBROUTINE 'TEST2'
82. .
83. REJECT
AP-40· INTEL MCS-48 KEYBOARD/DISPLAY APPLICATION NOTE APPENDIX

LOC OBJ SEG SOURCE STATEMENT

84 'INPUT-FOUR HIGH-ORDER BITS USED AT INPUTS FROM THE KEYBOARD RETURN LINES
85 . ASSUMES THAT A KEY DOWN IN THE CURRENTLY ENABLED ROW WOULD RETURN
86 : A LOW LEVEL
87 : IN THIS CASE, BIT4 RETURN THE LEFTMOST COLUMN, BIT4 THE RIGHTMOST
88 : THE HIGH-ORDER BITS ARE USED SO THAT IF AN OFF-CHIP DECODER IS USED
89 : TO ENABLE UP TO 16 CHARACTERS, FOR EXAMPLE, IT COULD BE WRITTEN BY
90 : THE LOW ORDER BITS OF THE SAME PATTERN.
91 : NOTE ALSO THAT IF A SIXTEEN KEY MATRIX WERE ELECTRICALLY ORGANIZED
92 : IN A 20• ARRAY, ONLY TWO RETURN LINES WOULD BE NEEDED.
93 : (IN THIS CASE, PERHAPS TO AN 'Y COULD BE USED FOR INPUT BITS)
94 : PULL-UP RESISTORS ON THE RETURN LINES MIGHT BE IN ORDER IF THERE IS ANY
95 : POSSIBILITY OF A HIGH-IMPEDANCE CONDUCTIVE PATH THROUGH THE SWITCH WHEN
96 : IT IS SUPPOSED TO BE 'OPEN'
97 : (THIS PHENOMENON HAS ACTUALLY BEEN OBSERVED!) .
98 :
99 :
100 : THE DRIVERS USED IN THE PROTOTYPE WERE ALL NON-INVERTING IN THAT
101 : A HIGH LEVEL ON AN OUTPUT LINE IS USED TO TURN A CHARACTER OR SEGMENT ON
102 : THERE ARE A TOTAL OF SEVEN 1/2 LINES LEFT OVER
103 :
104 : THE ALGORITHM FOR DRIVING THE DISPLAY USES A BLOCK OF INTERNAL RAM
105 : AS DISPLAY REGISTERS, WITH ONE BYTE-corresponding TO EACH CHARACTER OF THE
106 : DISPLAY, THE EIGHT BITS OF EACH BYTE CORRESPOND TO THE SEVEN SEGMENTS & UP
107 : OF EACH CHARACTER. IF AN EXTERNAL ENCODER IS USED (SUCH AS A FOUR-BIT TO
108 : SEVEN-SEGMENT ENCODER OR A ROM FOR TRANSLATING ASCII TO
109 : SIXTEEN-SEGMENT "STARBUCK" DISPLAY PATTERNS), THE TABLE ENTRIES WOULD HOLD
110 : THE CHARACTER CODES. (IN THE FORMER CASE, AN UNUSED BIT COULD BE USED TO
111 : ENABLE THE D P)
112 :
113 : THIS WRITING CHARACTERS TO THE DISPLAY FROM THE BACKGROUND PROGRAM
114 : REALLY ENTAILS WRITING THE APPROPRIATE SEGMENT
115 : PATTERNS TO A DISPLAY REGISTER- THE ACTUAL OUTPUTTING IS AUTOMATIC
116 : THE LEFTMOST CHARACTER CORRESPONDS TO THE LAST BYTE OF THE DISPLAY
117 : REGISTERS, AND IS ACQUIRED BY NEXTPL=6 (SEE SOURCE), THE RIGHTMOST
118 : CHARACTER IS THE FIRST DISPLAY BYTE, WHEN NEXTPL=1.
119 : UTILITY SUBROUTINES ARE INCLUDED HERE TO TRANSLATE FOUR BIT NUMBERS TO HEX
120 : DIGIT PATTERNS, AND WRITE THEM INTO THE DISPLAY REGISTERS SEQUENTIALLY
121 : (EITHER FILLING FROM THE LEFT- H P. CALCULATOR STYLE OR FROM THE
122 : RIGHT- TI STYLE, SUBROUTINES WISP AND RENTRY, RESPECTIVELY)
123 :
124 : THE KEYBOARD SCANNING ALGORITHM SHOWN HERE REQUIRE A KEY BE DOWN FOR
125 : SOME NUMBER OF COMPLETE DISPLAY SCANS TO BE ACKNOWLEDGED. SINCE IT IS
126 : INTENDED FOR 'ONE-FINGER' OPERATION, TWO-KEY ROLLOVER/N-KEY LOCKOUT HAS
127 : BEEN IMPLEMENTED, HOWEVER, MODIFICATIONS WOULD BE POSSIBLE TO ALLOW FOR
128 : EXAMPLE, ONE KEY IN THE MATRIX TO BE USED AS A SHIFT KEY OR CONTROL KEY
129 : TO BE HELD DOWN WHILE ANOTHER KEY IN THE MATRIX IS PRESSED (SEE NOTE WITHIN
130 : THE BODY OF THE LISTING.)
131 : EJECT

All mnemonics copyrighted © Intel Corporation 1976.

1-32
122: (Be aware that no more than two keys can ever be down unless diodes
123: are placed in series with all of the switches— certainly not the case for el.
124: cheapo keyboards— because some combinations of three keys down will result
125: in a 'phantom' fourth key being perceived.
126: The phantom key would be the fourth corner when three keys forming
127: a rectangular pattern (in the X-Y key matrix) are down.
128: If diodes are placed in the scanning array, considerations must be made
129: about how the diode voltage drop will affect input logic levels
130:
131: When a debounced key is detected, the number of its position in the key
132: matrix (left-to-right, bottom-to-top, starting from 00) is placed into
133: RAM location 'KBDUF' an input subroutine then need only read this location
134: repeatedly to determine when a key has been pressed when a key is detected.
135: A special code byte should be written back into 'KBDUF' to prevent
136: repeated detections of the same key.
137: The routine 'KBDIN' demonstrates a typical input protocol, along with a method
138: for translating a key position to its associated significance by accessing
139: Table 'Legends' in ROM.
140:
141: reject
### INITIAL EQUATES TO DEFINE SYSTEM CONFIGURATION

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>152</td>
<td></td>
<td></td>
<td><strong>------------------</strong></td>
</tr>
<tr>
<td>153</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>154</td>
<td></td>
<td></td>
<td>INITIAL EQUATES TO DEFINE SYSTEM CONFIGURATION</td>
</tr>
<tr>
<td>155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>156</td>
<td></td>
<td></td>
<td><strong>------------------</strong></td>
</tr>
<tr>
<td>157</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0010 158 PDIGIT EQU BUS -1 USED TO ENABLE CHARACTERS AND STROBE ROWS OF KEYBOARD
0008 159 PSEGNT EQU F1 -1 USED TO TURN ON SEGMENTS OF CURRENTLY ENABLED DIGIT
0009 160 PINPUT EQU P2 -1 PORT USED TO SCAN FOR KEY CLOSURES
161 | | | NOTE THAT THIS PORT ALLOCATION USES THE HIGHER |
162 | | | CURRENT SOURCING ABILITY OF THE BUS TO SWITCH ON THE |
163 | | | (DIGIT DRIVERS, AND LEAVES P22-P20 FREE FOR USING |
164 | | | AN 8243 PORT EXPANDER IN THE SYSTEM ) |
165 | | |                   |
0000 166 POSLOG EQU 00H -1 DEFINES WHETHER OUTPUT LINES ARE ACTIVE HI OR LOW |
00FF 167 NEGLOG EQU 0FFH -1 FOR DRIVING CHARACTERS AND SEGMENT PATTERNS |
0000 168 CHFQIL EQU POSLOG -1 NUMBER OF SEGMENTS IN DISPLAY |
0000 170 SEQFOL EQU POSLOG -1 ROWS OF KEYS LESS THAN OR EQUAL TO CHARNO |
00F0 171 INFMASK EQU 0FH -1 DEFINES BITS USED AS INPUT |
172 | | |                   |
0008 173 CHARNO EQU 8 -1 NUMBER OF DIGITS IN DISPLAY |
0004 174 NROWS EQU 4 -1 ROWS OF KEYS LESS THAN OR EQUAL TO CHARNO |
0004 175 NCOLS EQU 4 -1 LESSER DIMENSION OF KEYBOARD MATRIX |
176 | | |                   |
FF00 177 TICK EQU -10H -1 DEPENDS ON SUCCESSIVE SCANS BEFORE KEY CLOSURE ACCEPTED |
178 | | |                   |
0000 179 BLANK EQU 00H -1 CODE TO BLANK DISPLAY CHARACTERS |
180 | | |                   |
181 | | | (WOULD BE 20H IF ASCII DECODING ROM USED OR 0FH IF  |
182 | | | $7447$-TYPE SEVEN-SEGMENT DECORATOR EXTERNAL TO 8748) |
184 | | |                   |
000F 183 ENCMAS EQU 0FH -1 SELECTS WHICH BITS ARE RELEVANT TO ENCC SUBROUTINE |
184 | | |                   |
185 | | | EJECT |

---

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ SEQ SOURCE STATEMENT

186 ,********************************************************************
187 :
188 , BANK 0 REGISTERS USED
189 :
190 ; POINTERS USED FOR INDIRECT RAM ACCESSING.
0000 191 PNTR0 EQU R0
0001 192 PNTR1 EQU R1
0007 193 NEXTPL EQU R7 ; USED TO KEEP TRACK OF CHARACTER POSITION BEING
194 ; WRITTEN INTO
195 :
196 ; ********************************************************************
197 :
198 ; BANK 1 REGISTER ALLOCATION
199 :
200 ; PNTR0 EQU R0 (ALREADY DEFINED)
201 ; PNTR1 EQU R1
0002 202 ASAVE EQU R2 ; HOLDS ACCUMULATOR VALUE DURING SERVICE ROUTINE
0004 203 ROTPAT EQU R4 ; USED TO HOLD INPUT PATTERN BEING ROTATED THROUGH CY
0005 204 ROTCNT EQU R5 ; COUNTS NUMBER OF BITS ROTATED THROUGH CY
0006 205 LASTKY EQU R6 ; HOLDS KEY POSITION OF LAST KEY DEPRESSION DETECTED
0007 206 CURDIG EQU R7 ; HOLDS POSITION OF NEXT CHARACTER TO BE DISPLAYED
207 :
208 ;********************************************************************
209 :
210 ; DATA RAM ALLOCATION
211 :
0020 212 NEXTPTS EQU 32 ; KEEPS TRACK OF SUCCESSIVE READS OF SAME KEYSKROKE
0021 213 KEYLOC EQU 33 ; INCREMENTED AS SUCCESSIVE KEY LOCATIONS SCANNED
0022 214 KBDBUF EQU 34 ; CARRIES POSITION OF DEBOUNDED KEY FROM REFRESH ROUTINE
215 : \ BACK TO BACKGROUND PROGRAM
0023 216 RDELAY EQU 35 ; NON-ZERO WHEN DISPLAY IN PROGRESS
217 ;
218 ; THE LAST (CHARNO) REGISTERS HOLD THE DISPLAY SEGMENT PATTERNS
219 ;
0037 220 SEGMAP EQU (63-CHARNO) ; BASE OF REGISTER ARRAY FOR DISPLAY PATTERNS
221 : \ (COULD BE ANYWHERE IN INTERNAL RAM)
222 ;
223 ;********************************************************************
224 :
225 ; NOTE THAT LASTKY, CURDIG, AND F1 RETAIN STATUS INFORMATION FROM
226 ; ONE INTERRUPT TO THE NEXT. ALL OTHER REGISTERS MAY BE USED IN
227 ; THE USER'S OWN INTERRUPT SERVICING ROUTINE
228 ;
229 ;********************************************************************
230 ;
231 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ SED SOURCE STATEMENT

222 , 223 ;******************************************************************************
224 ;
225 ORG 000H
226 JMP INIT
227 ;
228 ;
229 ;******************************************************************************
240 ;
241 ORG 007H
242 ;
243 TIINT TIMER INTERRUPT SUBROUTINE
244 ; CALL MADE TO LOC 007H WHEN TIMER TIMES OUT
245 ; TIMER CAN BE RE-INITIALIZED AT THIS POINT IF DESIRED.
246 ; USED HERE TO CAUSE THE DISPLAY REFRESH AND KEY SCAN ROUTINES TO
247 ; BE CALLED PERIODICALLY.
248 ;
249 TIINT: SEL RB1
250 MOV ASAVE,A
251 MOV R,#TICK
252 ;
253 ;******************************************************************************
254 ;
255 ; THE USER'S OWN TIMER INTERRUPT ROUTINE (IF IT EXISTS) COULD
256 ; BE PLACED AT THIS POINT
257 ;
258 ;******************************************************************************
259 ;
260 CALL REFRESH ;CAUSE DISPLAY TO BE UPDATED
261 ;
262 ; THE COMPLETE INTERRUPT ROUTINE SHOULD BE COPIED HERE
263 ; TO SAVE A FULL LEVEL OF SUBROUTINE NESTING
264 ; IT WAS WRITTEN AS A SUBROUTINE HERE FOR THE SAKE OF CLARITY.
265 ;
266 ;******************************************************************************
267 ;
268 ;TIERT TIMER INTERRUPT RETURN CODE- RESTORES HCC VALUE
269 TIER: MOV A,ASAVE
270 RETR
271 ;
272 *EJECT
LOC OBJ SEQ SOURCE STATEMENT

273:******************************************************************************
274:REFRESH SUBROUTINE TO MULTIPLEX SEVEN-SEGMENT DISPLAYS
275:  EACH CALL CAUSES THE NEXT CHARACTER TO BE DISPLAYED.
276:  ACCORDING TO THE CONTENTS OF THE SEGMAP REGISTER ARRAY
277:  REFRESH SHOULD BE CALLED AT LEAST EVERY 50 MSEC OR SO
278:******************************************************************************
279:
0010 2300 280 REFRESH: MOV A, #BLANK XOR SEGPOL
0012 39 281 OUTL PSGMNT.A  ;WRITE BLANK PATTERN TO SEG DRIVERS
0013 2357 282 KEFR1: MOV A, #CHRSTB  ;LOOK UP DIGIT ENABLE PATTERN
0015 6F 283 ADD A, CURDIG  ;ADD CURDIG DISPLACEMENT
0016 E3 284 MOVX A, @R0  ;ENABLE ONE BIT OF ACCUMULATOR
0017 02 285 OUTL PDIGIT.A  ;ENERGIZE CHARACTER
286:
287:
0018 2337 288 MOV A, #SEGMAP  ;LOAD BASE OF REGISTER ARRAY
001A 6F 289 ADD A, CURDIG  ;ADD CURDIG DISPLACEMENT
001B A9 290 MOV @PNTR1.A
001C F1 291 MOV A, @PNTR1  ;LOAD ACC W/ NEXT SEGMENT PATTERN
001D 39 292 OUTL PSGMNT.A  ;ENABLE APPROPRIATE SEGMENTS
293:
294:******************************************************************************
295:  THE NEXT CHARACTER IS NOW BEING DISPLAYED.
296:  THE KEYBOARD SCAN ROUTINE IS INTEGRATED INTO THE DISPLAY SCAN.
297:  WITH THE CURRENT ROW ENERGIZED, CHECK IF THERE ARE ANY INPUTS
298:******************************************************************************
299:
001E 8821 300 SCAN: MOV PNTR0, KEVLOC  ;SET POINTER FOR SEVERAL KEVLOC REFERENCES
0020 0A 301 IN A, PINPUT  ;LOAD ANY SWITCH CLOSURES
302:
303:******************************************************************************
304:  THIS BLOCK OF CODE IS NOT NEEDED BY THE KEYBOARD SCAN LOGIC
305:  HOWEVER, ITS INCLUSION WOULD SPEED THINGS UP A BIT BY
306:  SKIPPING OVER ROWS IN WHICH NO KEYS ARE DOWN.
307:  IT WAS OMITTED HERE TO CONSERVE ROM SPACE.  BUT MIGHT BE
308:  RESTORED IF VERY LARGE KEYBOARDS (ESPECIALLY THOSE WITH EIGHT
309:  KEYS PER ROW) ARE TO BE USED WITH THIS ALGORITHM
310:******************************************************************************
311:
312:
313:
314:
315:
316:
317:
318:
319:******************************************************************************
320:  IF THIS CODE IS USED, SUBSTITUTE THE 'JC SCANS' FOUR LINES
321:  HENCE WITH 'JNC SCANS' TO ACCOMMODATE THE INVERTED POLARITY
322:******************************************************************************
323:leanor
LOC OBJ   SED  SOURCE STATEMENT

324 ;******************************************************************
325 ; ROTATE BITS THROUGH CY WHILE INCORRECTING KEYLOC
326 ;******************************************************************
327 ,
0021 B044  328 SCANL: MOV R0, #NOLCS ;SET UP FOR <NOLCS> LOOPS THROUGH 'NXTLOC'
0023 F7   329 NXTLOC: RLC A
0024 FC   330 MOV R0, #Rotpat A ;SAVE SHIFTED BIT PATTERN
0025 F63F  331 JC SCAN5 ;ONE BIT IN CY INDICATES KEY NOT DOWN
0026 8921  333 ;******************************************************************
0027 8004  334 AT THIS POINT IT HAS JUST BEEN DETERMINED THAT THE VALUE
0028 8923  335 OF KEYLOC IS THE POSITION OF A KEY WHICH IS NOW DOWN
0029 F7    336 THE FOLLOWING CODE DEBOUNCES THE KEY, ETC.
002A FC    337 IF MODIFICATIONS TO THE KEYBOARD LOGIC, I.E. THE INCLUSION
002B 8004  338 OF A SHIFT, CONTROL, OR MODE KEY IN THE KEY MATRIX ITSELF)
002C F7    339 ARE DESIRED, THEY SHOULD BE MADE AT THIS POINT, BEFORE
002D 8923  340 THE DEBOUNCE LOGIC BEGINS (FOR EXAMPLE: AT THIS POINT
002E 8004  341 KEYLOC COULD BE COMPARED AGAINST THE POSITION OF THE MODE
002F F7    342 KEY, AND IF THEY MATCH SET SOME FLAG BIT AND JUMP TO
0030 8923  343 LABEL 'SCANS' OR, BY COMPARE KEYLOC AGAINST THE LAST
0031 8004  344 KEY DEBOUNCED, IMMEDIATE TWO-KEY ROLLOVER COULD BE
0032 F7    345 IMPLEMENTED.
0033 8004  346 ;******************************************************************
0034 8921  348 ,
0035 8004  349
0036 F7   350 CLR F1 ;MARK THAT AT LEAST ONE KEY WAS DETECTED
0037 8004  351 CPL F1 ;IN THE CURRENT SCAN
0038 F7    352 ;******************************************************************
0039 8921  354 A KEYSTROKE WAS DETECTED FOR THE CURRENT COLUMN, ITS
0040 8004  355 POSITION IS IN REGISTER KEYLOC. SEE IF SAME KEY SENSED LAST CYCLE
0041 F7    356 ;******************************************************************
0042 8004  357
0043 8004  358 MOV A, #Patro ;Patro STILL HOLDS #KEYLOC
0044 F0    359 XCH A, Lastky
0045 8004  360 XRL A, Lastky
0046 8820  361 MOV Patro, #Repts ;PREPARE TO CHECK AND/OR MODIFY REPEAT COUNT
0047 C634  362 JC Scan5
0048 634

All mnemonics copyrighted © Intel Corporation 1976.
LOC  OBJ  SED  SOURCE STATEMENT

365: **-------------------------------------------------------------------**
366: A DIFFERENT KEY WAS READ ON THIS CYCLE THAN ON THE PREVIOUS CYCLE.
367: SET NREPTS TO THE DEBOUNCE PARAMETER FOR A NEW COUNTDOWN.
368: **-------------------------------------------------------------------**
369: 

0034 F03 380: SCAN5: MOV  @PNTR2, @DESBCE
0035 C63F 381: JZ  SCAN5  IF ALREADY ZERO
0037 07 382: DEC  A  INDICATE ONE MORE SUCCESSIVE KEY DETECTION
0038 A0 383: MOV  @PNTR0, A
0039 963F 384: JNZ  SCAN5  IF DECREMENT DOES NOT RESULT IN ZERO
003B FE 385: MOV  @LASTKY
003C B022 386: MOV  PNTR0, KEYBUF
003E A0 387: MOV  @PNTR0, A  TO MARK NEW KEY CLOSURE
388:
003F B021 389: SCAN5: MOV  PNTR0, #KEYLOC
0041 10 390: INC  @PNTR0
0042 FC 391: MOV  @ROTPAT
0043 ED25 392: DNZ  ROTCH, NXTLOC
393:
394:
0045 EF57 395: SCAN6: DNZ  CURDIG, SCAN9
396:
397: REJECT

All mnemonics copyrighted © Intel Corporation 1976.
THE FOLLOWING CODE SEGMENT IS USED BY THE KEYBOARD SCANNING ROUTINE
IT IS EXECUTED ONLY AFTER A REFRESH SEQUENCE OF ALL
THE CHARACTERS IN THE DISPLAY IS COMPLETED

0047 BF08  MOV  CURDIG,#CHARNO
0049 B800  MOV  @PNTR0,#0 ;PNTR0 STILL CONTAINS #KEYLOC
004B 764F  JF1  SCAN8 ;JUMP IF ANY KEYS WERE DETECTED
004D BEFF  MOV  LASTKY,#OFFH ;CHANGE LASTKY WHEN NO KEYS ARE DOWN
004F A5    SCAN9:  CLR  F1
11

THE NEXT CODE SEGMENT IS THE INTERRUPT-DRIVEN PORTION OF THE 'DELAY'
UTILITY IT DECREASES RAM LOCATION 'RDELAY' ONCE PER DISPLAY SCAN
IF 'RDELAY' IS NOT ALREADY ZERO

0050 B923  MOV  PNTR1,#RDELAY
0052 F1    MOV  (PNTR1,#RDELAY)
0053 057   JZ  SCAN9
0055 07    DEC  A
0056 A1    MOV  @PNTR1,A
0057 83    SCAN9: RET

CHRSTB IS THE BASE FOR THE PATTERNS TO ENABLE ONE-OF-CHARNO CHARACTERS.
0057 420  CHRSTB  OR  (A-1) AND #OFFH
0059 01    DB  (00000000 XOR CHRPOL)
0059 00    DB  (00000001 XOR CHRPOL)
0059 04    DB  (00000100 XOR CHRPOL)
0059 08    DB  (00001000 XOR CHRPOL)
005C 10    DB  (00001001 XOR CHRPOL)
005E 20    DB  (00010000 XOR CHRPOL)
005E 3F    DB  (01000000 XOR CHRPOL)
005F 80    DB  (10000000 XOR CHRPOL)
438 #ECT
LOC 0E1  SED  SOURCE STATEMENT

479 .INIT Initializes processor registers

0060 DS 440 INIT SEL RB1
0061 BF00 441 MOV CORDIG, CHARNO
0062 B022 442 MOV FNTRB, #BE0UP
0065 B0FF 443 MOV @FNTRB, #FFH
0067 B021 444 MOV FNTRB, #KEYLOC
0069 B000 445 MOV @FNTRB, #0
0068 23F0 446 MOV A, #INPSK
006D 3A 447 OUTL "INPUT.A" ; Set bidirectional input lines
006E C5 448 SEL RB0
006F 149E 449 CALL CLEAR ; Utility for setting initial display registers.
0071 65 450 CLR FI
0072 23F0 451 MOV A, #TICK ; Load interrupt rate value
0074 62 452 MOV T.R
0075 55 453 STR T
0076 25 454 EN TONTI ; Enable timer interrupts
455 ;
456 ;
457 :******************************************************************************
458 :
459 ;ECHO Check for any new keystrokes detected
460 ; Translate each keystroke into a segment pattern
461 ; AND write it into the appropriate display register
462 ;
463 :******************************************************************************
464 :
0077 1493 465 ECHO CALL KBDIN ; Get next keystroke
0079 B281 466 JBE5 FKEY ; Jump if key in right-hand column
467 ; Since the ACC is used by ENACC and RENTRY, its contents must
468 ; be processed or saved before ENACC is called
0078 149A 469 CALL ENACC ; Form appropriate segment pattern
007D 1408 470 CALL RENTRY ; Write pattern into display registers
007F 0477 471 JMP ECHO ; Loop indefinitely

472 ;
0081 2400 473 FKEY JMP FUNCNT ; Jump to off-page code to call demo routine
474 ;
475 #EJECT
The following subroutines implement the utilities commonly used for most keyboard/display applications. They could be used exactly as shown here or adapted for special cases.

KBDIN KEYBOARD INPUT SUBROUTINE.

KBDIN should be used to interface the user's background program with the interrupt driven keyboard scanner. It returns only after a new keystroke has been detected and debounced. The encoded value of key (rather than its position in switch matrix) is returned in the accumulator.

KBDIN MOV PNT1, #KBDUF
KBDIN MOV #M, #MPT1
KBDIN MOV #M, #KBDIN
KBDIN ADD #LEGND
KBDIN MOV #A, #KBDIN
KBDIN ADD(BASE OF KEY ENCODING TABLE)
KBDIN MOV #A, #H
KBDIN MOV PTR #KBDUF

LEGND IS THE BASE FOR TABLE SHOWING KEY MATRIX SIGNIFICANCE FOR THE KEYBOARD USED IN THE PROTOTYPE.

LEGND IS THE BASE FOR TABLE SHOWING KEY MATRIX SIGNIFICANCE FOR THE KEYBOARD USED IN THE PROTOTYPE.

NOTE THAT BIT 4-BIT 7 MAY BE USED TO ENCODER KEYS. IN THIS CASE:

- BIT 4 INDICATES REGULAR DECIMAL DIGITS.
- BIT 5 INDICATES RIGHT-COLUMN FUNCTION KEYS.
- BIT 6 INDICATES PUNCTUATION MARKS (AND \).

LEGND EQU ($ AND $0FF) USE LOW ORDER BITS AS TABLE INDEX

LEGND EQU ($ AND $0FF) USE LOW ORDER BITS AS TABLE INDEX

LOC 081 SEG SOURCE STATEMENT

476.***************************************************************
477.
478. THE FOLLOWING SUBROUTINES IMPLEMENT THE UTILITIES COMMONLY USED FOR
479. MOST KEYBOARD/DISPLAY APPLICATIONS.
480. THEY COULD BE USED EXACTLY AS SHOWN HERE OR ADAPTED FOR SPECIAL CASES.
481.
482.***************************************************************
483.:
484.:
485.:
486.:
487.:
488.:
489.:
490.:
491.:
492.:
493.:
494.:
495.:
496.:
497.:
498.:
499.:
500.:
501.:
502.:
503.:
504.:
505.:
506.:
507.:
508.:
509.:
510.:
511.:
512.:
513.:
514.:
515.:
516.:
517.:
518.:
519.:
520.:
521.:
522.:
523.:
524.:
525.:

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ SE0 SOURCE STATEMENT

526.**************************************************************
527
528: CLEAR WRITES 'BLANK' CHARACTERS INTO ALL DISPLAY REGISTERS.
529: 
530: FILL WRITES SEGMENT PATTERN NON IN ACC INTO ALL DISPLAY REGISTERS
531: 
532: 2200 521 CLEAR MOV A,BLNK X0R SEGPOL
533: 8808 532 FILL MOV PNTR1,#SEGARP+1
534: 8808 533 MOV NEXTPL,#CHARNO
535: 8808 534 CLR1 MOV @PNTR1,A
536: 8805 535 INC PNTR1
537: 8808 536 MOV NEXTPL,CLR1
538: 8808 537 MOV NEXTPL,#CHARNO
539: 540:**************************************************************
541: 
542: PRINT SUBROUTINE TO COPY A STRING OF BIT PATTERNS FROM ROM TO THE
543: DISPLAY REGISTERS. STRING STARTS AT LOCATION POINTED TO BY PNTR0.
544: CONTINUES UNTIL AN ESCAPE CODE (OFFH) IS REACHED.
545: NOTE THAT THE CHARACTER STRING PUT OUT MUST BE LOCATED ON THE SAME
546: PAGE AS THIS SUBROUTINE, SINCE SAME-PAGE MOVES ARE USED.
547: PRINT IN TURN CALLS EITHER SUBROUTINE 'WDISP' OR 'ENTRY.'
548: TO ACTUALLY EFFECT WRITING INTO THE DISPLAY REGISTERS.
549: 
550: F8 549 PRINT: MOV A,PNTR0
551: 0B 550 MOVP A,AH
552: 08 551 JZ PNTR1
553: 40 552 CALL WDISP
554: 18 553 INC PNTR0
555: 04 554 JMP PRINT
556: 83 555 RET
557: 558:**************************************************************
559: 
560: JOHN ARRAY HOLDS THE BIT PATTERNS FOR THE LETTERS 'JOHN' (SEE 'TEST2')
561: (NOTE THAT 'OHN' IS WRITTEN IN LOWER CASE LETTERS)
562: 
563: 05 562 JOHN EQU $ AND OFFH
564: 1E 563 DB 00111110B XOR SEGPOL
565: 5C 564 DB 01011100B XOR SEGPOL
566: 74 565 DB 01101000B XOR SEGPOL
567: 54 566 DB 01010100B XOR SEGPOL
568: 00 567 DB 00
569: 569 RET
LOC OBJ SOURCE STATEMENT

570 : ****************************************************************************************************
571 :
572 : ENCACC ENCODES LSNIBBLE OF ACC INTO HEX BIT PATTERN INTO ACC
008A 530F 573 ENCACC ANL A, #ENCMSK
008C 02C0 574 ADD A, #DPATS
008E A3 575 MOV P A, #A
008F 83 576 RET
577 : DPATS IS THE BASE FOR THE TABLE OF SEGMENT PATTERNS FOR THE BASIC
578 : DIGITS. HERE THE FULL HEX SET (0-F) IS INCLUDED.
579 : FOR MANY USER APPLICATIONS, THE CHARACTER SET MAY BE AMENDED OR AUGMENTED
580 : TO INCLUDE ADDITIONAL SPECIAL PURPOSE PATTERNS.
581 : FORMAT IS PGFEDCBA IN STANDARD SEVEN-SEGMENT ENCODING CONVENTION
582 : WHERE P REPRESENTS THE DECIMAL POINT
00C8 583 DPATS EOU A AND 8FFH
00C8 3F 584 DB 00111111B XOR SEGPO
00C8 06 585 DB 00000110B XOR SEGPO
00C8 5B 586 DB 01011011B XOR SEGPO
00C8 4F 587 DB 01011111B XOR SEGPO
00C8 66 588 DB 01100110B XOR SEGPO
00C8 6D 589 DB 01101011B XOR SEGPO
00C8 76 590 DB 01111010B XOR SEGPO
00C8 07 591 DB 00000111B XOR SEGPO
00C8 7F 592 DB 01111111B XOR SEGPO
00C8 67 593 DB 01100111B XOR SEGPO
00C8 77 594 DB 01110111B XOR SEGPO
00C8 7C 595 DB 01111000B XOR SEGPO
00C8 39 596 DB 00110001B XOR SEGPO
00C8 5E 597 DB 01011110B XOR SEGPO
00C8 79 598 DB 01110010B XOR SEGPO
00CF 71 599 DB 01110001B XOR SEGPO
600 :
601 : *****************************************************
602 :
603 : WDISP WRITES BIT PATTERN NOW IN ACC INTO NEXT CHARACTER POSITION
604 : OF THE DISPLAY (NEXTPL). ADJUSTS NEXTPL POINTER VALUE.
605 : RESULTS IN DISPLAY BEING FILLED LEFT TO RIGHT, THEN RESTARTING
00D0 A9 606 WDISP MOV PNTRL.A
00D1 FF 607 MOV A, NEXTPL
00D2 0337 608 ADD A, #SEGMAP
00D4 29 609 XCH A, PNTRL
00D5 A1 610 MOV #PNTRL.A
00D6 EFDA 611 DJNZ NEXTPL, WDISP1
00D8 BF08 612 MOV NEXTPL, #CHARNO
00D9 83 613 WDISP1 RET
614 :
615 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBI  SENO  SOURCE STATEMENT

616 :********************************************************************
617
618  ENTRY SUBROUTINE TO ENTER ACC CONTENTS INTO THE RIGHTMOST DIGIT
619  AND SHIFT EVERYTHING ELSE ONE PLACE TO THE LEFT
000B 8938  620  ENTRY:  MOV  PTR1,#SEGMAP+1
000D 8F08  621  MOV  NEXTFL,#CHAR0
000F 21  622  RENT1  XOR  A,#PTR1
000E 19  623  INC  PTR1
0001 EFDF  624  DNZ  NEXTFL:RENT1
0002 BF08  625  MOV  NEXTFL,#CHAR0:POINT TO LEFTMOST CHARACTER
0005 83  626  RET
627
628 :********************************************************************
629
630  :OPADD TOGGLE DECIMAL POINT IN LAST CHARACTER DISPLAY CHARACTER
631 :OPADD TOUGLES DECIMAL POINT IN THE CHARACTER POINTED TO BY THE ACC
632
0006 2381  633  POPADD  MOV  A,#01H  :SET INDEX TO RIGHMOST POSITION
0008 8337  634  OPADD  ADD  A,#SEGMAP  :ACCESS DISPLAY REGISTER FOR DESIRED PLACE
000A 89  635  MOV  PNTR1,A
000B F1  636  MOV  A,#PNTR1
000C D300  637  XRL  A,#80H
000E F1  638  MOV  #PNTR1,A
000F 82  639  RET
640
641 :********************************************************************
642
643  :HOLD  SUBROUTINE CALLED WHEN KEY IS KNOWN TO BE DOWN.
644  : WLL NOT RETURN UNTIL KEY IS RELEASED:
0008 D5  645  HOLD:  SEL  R81
000F 1E  646  MOV  A,#LASCY  :LASTKEY=OFFH IFF NO KEYS DOWN
0002 C5  647  SEL  R80
0003 37  648  CPL  A
000F 96F0  649  INC  HOLD
000F 83  650  RET
651
652 :********************************************************************
653
654  :DELAY SUBROUTINE HANGS UP FOR THE NUMBER OF COMPLETE DISPLAY SCANS EQUAL
655  TO THE CONTENTS OF THE ACCUMULATOR WHEN CALLED:
0007 8923  656  DELAY:  MOV  PNTR1,#DELAY
0009 A1  657  MOV  @PNTR1,A
000A F1  658  DELAY1:  MOV  #PNTR1
000B 96F9  659  JNZ  DELAY1
000D 83  660  RET
661  REJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ      SEG       SOURCE STATEMENT

0100  662 095   LOOH
       663 :
       664 :************************************************************
       665 : THE CODE ON THIS PAGE IS FOR DEMONSTRATION PURPOSES ONLY-
       666 : I TRULY DOUBT WHETHER ANY END USERS WOULD LIKE TO SEE A NAME
       667 : POPPING UP ON THEIR CALCULATOR SCREENS
       668 : HOWEVER THE CODE SHOWN HERE DOES INDICATE HOW THE UTILITY SUBROUTINES
       669 : INCLUDED HERE COULD BE ACCESSED
       670 : THE ROUTINES THEMSELVES ARE CALLED WHEN ONE OF THE FOUR BUTTONS
       671 : ON THE RIGHT-HAND SIDE OF THE PROTOTYPE KEYBOARD IS Pressed.
       672 :
       673 : ************************************************************
       674 : FUNCTN ROUTINE TO IMPLEMENT ONE OF FOUR DEMO UTILITIES, ACCORDING
       675 : TO WHICH OF THE FOUR FUNCTION KEYS WAS Pressed.
       0100 1212
       0102 320E  676  1B0  FUNCT1
       0104 520A
       677 JEB2  FUNCT2
       681 :
       0106 1AE6  682  FUNCT4. CALL RDPADD
       0108 0477
       683 JMP ECHO
       684 :
       010A 342E
       685  FUNCT3. CALL TEST3
       010C 0477
       686 JMP ECHO
       687 :
       010E 3424
       688  FUNCT2: CALL TEST2
       0110 0477
       689 JMP ECHO
       690 :
       0112 3416
       691  FUNCT1: CALL TEST1
       0114 0477
       692 JMP ECHO
       693 :
       694 :************************************************************
       695 :
       0116 8F08
       696  TEST1. CODE SEGMENT TO FILL DISPLAY REGISTERS WITH DIGITS DOWN TO '1'
       0118 8008
       697  MOV NEXTPL, CHARNO
       011A FF
       698  MOV PNTRO, CHARNO :SET FOR EIGHT LOOP REPLICATIONS
       011C 1400
       699  TST11: MOV R-NEXTPL
       011E 140A
       700  CALL ENCHCC
       0120 1400
       701  CALL WDISP
       0122 E81A
       702  CALL PNTRO.TST11: COPY NEXT DIGIT INTO DISPLAY REGISTERS
       0124 4B08
       703  MOV NEXTPL, CHARNO
       0126 83
       704 RET
       705
       706 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
**LOC OBJ**

```
707 : **********************************************/
708 :
709 : TEST2 WRITES THE SEGMENT PATTERN FOR 'JOHN' ONTO THE DISPLAY.
710 : WAITS FOR A WHILE AND THEN CLEARS THE DISPLAY
0124 6885
0126 148
0128 234
0129 14F
012C 049E
012D 0123 0126 012A 012B 012C
711 TEST2: MOV PNTR0.@JOHN
712 CALL PRINT
713 MOV A #100, SCAN DISPLAY FOR 100 CYCLES
714 CALL DELAY
715 JMP CLEAR
716 :
717 : **********************************************/
718 :
719 : TEST3 SUBROUTINE TO FILL DISPLAY WITH DASHES
720 : JUMPS INTO SUBROUTINE 'CLEAR'
721 : AS SOON AS THE KEY IS RELEASED.
012E 2340
0130 14F0
0132 14F0
0134 049E
0135 722 TEST2 MOV A, #01000000 XOR SEGPOL, PATTERN FOR '-'
0136 723 CALL FILL
0138 724 CALL HOLD
013A 725 JMP CLEAR
726 :
727 : **********************************************/
728 :
729 END:
```

**USER SYMBOLES**

```
ASAVE 0002 BLANK 0000 CHRN0 0008 CHPOL 0000 CHSIB 0057 CLEAR 009E CLR1 00A4 CURDIG 0007
DEBCE 00E4 DELAY 00F7 DELAY1 00FA DPATS 00C0 DPADD 00E3 ECHO 0077 ENCCAC 009A ENCIBM 000F
FILL 00A0 FKET 0081 FUNCT0 0112 FUNCT2 018E FUNCT3 019A FUNCT4 0106 FUNCIN 0100 HOLD 00F0
INIT 00E0 Int MSK 00F0 JOHN 0085 YBOFB 0022 KBIN 00B3 KEYLOC 0021 LASTKY 0006 LEGNOS 00BE
NCOLS 0004 NEGLOG 00F NEXTPL 0007 NPEPTS 0028 NROWS 0004 NVTLOC 0023 PDIGIT 0010 PINPUT 0089
PNTR0 0000 PNTR1 0001 POSLOG 0000 PRINT 008B PFSNT1 0084 PSEGMENT 0008 RDELAY 0023 RDADD 00E6
PEPS 0012 PFRSH 0019 KENTR1 0010 PENTRY 0008 ROTPO 0095 ROTPAT 0004 SCAN 000E SCANT 0021
SCANS 0054 SCANN 003F SCANG 0045 SCANA 004F SCANH 0057 SEGMAP 0037 SEGPOL 0000 TEST1 0116
TEST2 0124 TEST3 0126 TICL 0000 TIFL 0007 TINET 000E TST1 011A WDISP 0000 WDISP 000A
```

**ASSEMBLY COMPLETE. NO ERRORS**
### IS15-11 ASSEMBLER SYMBOL CROSS REFERENCE, V2

<table>
<thead>
<tr>
<th>ASRM</th>
<th>282#</th>
<th>249</th>
<th>269</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLANK</td>
<td>179#</td>
<td>200</td>
<td>531</td>
</tr>
<tr>
<td>CHARNO</td>
<td>173#</td>
<td>220</td>
<td>463</td>
</tr>
<tr>
<td>CHRPOL</td>
<td>169#</td>
<td>429</td>
<td>430</td>
</tr>
<tr>
<td>CMSTB</td>
<td>262#</td>
<td>428#</td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td>449#</td>
<td>531#</td>
<td>715</td>
</tr>
<tr>
<td>CLR1</td>
<td>534#</td>
<td>536</td>
<td></td>
</tr>
<tr>
<td>CRDIG</td>
<td>206#</td>
<td>283</td>
<td>289</td>
</tr>
<tr>
<td>DENCE</td>
<td>178#</td>
<td>370</td>
<td></td>
</tr>
<tr>
<td>DELAY</td>
<td>656#</td>
<td>714</td>
<td></td>
</tr>
<tr>
<td>DELAY1</td>
<td>659#</td>
<td>659</td>
<td></td>
</tr>
<tr>
<td>IGARTS</td>
<td>574#</td>
<td>563#</td>
<td></td>
</tr>
<tr>
<td>DPADD</td>
<td>534#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECHO</td>
<td>465#</td>
<td>471</td>
<td>583</td>
</tr>
<tr>
<td>ENCHCC</td>
<td>469#</td>
<td>573#</td>
<td>700</td>
</tr>
<tr>
<td>ENDSK</td>
<td>193#</td>
<td>573</td>
<td></td>
</tr>
<tr>
<td>FILL</td>
<td>532#</td>
<td>723</td>
<td></td>
</tr>
<tr>
<td>FKEY</td>
<td>466#</td>
<td>473#</td>
<td></td>
</tr>
<tr>
<td>FNC1</td>
<td>678#</td>
<td>691#</td>
<td></td>
</tr>
<tr>
<td>FNC2</td>
<td>679#</td>
<td>688#</td>
<td></td>
</tr>
<tr>
<td>FNC3</td>
<td>680#</td>
<td>685#</td>
<td></td>
</tr>
<tr>
<td>FNC4</td>
<td>682#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FNCN1</td>
<td>472#</td>
<td>678#</td>
<td></td>
</tr>
<tr>
<td>HOLD</td>
<td>645#</td>
<td>649</td>
<td>724</td>
</tr>
<tr>
<td>INIT</td>
<td>236#</td>
<td>440#</td>
<td></td>
</tr>
<tr>
<td>INPMASK</td>
<td>171#</td>
<td>446</td>
<td></td>
</tr>
<tr>
<td>JOHN</td>
<td>562#</td>
<td>711</td>
<td></td>
</tr>
<tr>
<td>KDBUF</td>
<td>214#</td>
<td>386</td>
<td>442</td>
</tr>
<tr>
<td>KDBIN</td>
<td>463</td>
<td>490#</td>
<td>493</td>
</tr>
<tr>
<td>KEYLOC</td>
<td>213#</td>
<td>380</td>
<td>389</td>
</tr>
<tr>
<td>LASTKY</td>
<td>205#</td>
<td>359</td>
<td>380</td>
</tr>
<tr>
<td>LENO5</td>
<td>494#</td>
<td>508#</td>
<td></td>
</tr>
<tr>
<td>NCOLS</td>
<td>175#</td>
<td>328</td>
<td></td>
</tr>
<tr>
<td>NEGLOG</td>
<td>167#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEXIT1</td>
<td>153#</td>
<td>533</td>
<td>536</td>
</tr>
<tr>
<td>NFNEPTS</td>
<td>212#</td>
<td>361</td>
<td></td>
</tr>
<tr>
<td>NROWS</td>
<td>174#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NXLCN</td>
<td>329#</td>
<td>392</td>
<td></td>
</tr>
<tr>
<td>PDIGIT</td>
<td>158#</td>
<td>285</td>
<td></td>
</tr>
<tr>
<td>PINPUL</td>
<td>160#</td>
<td>381</td>
<td>447</td>
</tr>
<tr>
<td>PNTRB</td>
<td>191#</td>
<td>388</td>
<td>358</td>
</tr>
<tr>
<td>PNTR1</td>
<td>192#</td>
<td>290</td>
<td>291</td>
</tr>
<tr>
<td>POSLOG</td>
<td>166#</td>
<td>169#</td>
<td>170</td>
</tr>
<tr>
<td>PRINT</td>
<td>549#</td>
<td>555</td>
<td>712</td>
</tr>
<tr>
<td>PRNTR1</td>
<td>551#</td>
<td>556#</td>
<td></td>
</tr>
<tr>
<td>FSGNW</td>
<td>159#</td>
<td>261</td>
<td>292</td>
</tr>
<tr>
<td>RDELY</td>
<td>216#</td>
<td>417</td>
<td>636</td>
</tr>
<tr>
<td>RDPADD</td>
<td>633#</td>
<td>682</td>
<td></td>
</tr>
<tr>
<td>REF1</td>
<td>282#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFRESH</td>
<td>280#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RENTR1</td>
<td>622#</td>
<td>624</td>
<td></td>
</tr>
<tr>
<td>RENTRY</td>
<td>470#</td>
<td>620#</td>
<td></td>
</tr>
</tbody>
</table>

---

All mnemonics copyrighted © Intel Corporation 1976.
**ISIS-II ASSEMBLER SYMBOL CROSS REFERENCE: V2.0**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTCONT</td>
<td>204#</td>
</tr>
<tr>
<td>ROTPAT</td>
<td>203#</td>
</tr>
<tr>
<td>SCBH</td>
<td>300#</td>
</tr>
<tr>
<td>SCBH1</td>
<td>328#</td>
</tr>
<tr>
<td>SCBH2</td>
<td>362#</td>
</tr>
<tr>
<td>SCBH5</td>
<td>311 371 381 384 389#</td>
</tr>
<tr>
<td>SCBH6</td>
<td>395#</td>
</tr>
<tr>
<td>SCBH8</td>
<td>407 409#</td>
</tr>
<tr>
<td>SCBH9</td>
<td>395 419 423#</td>
</tr>
<tr>
<td>SEGHP</td>
<td>200# 200 322 606 620 634</td>
</tr>
<tr>
<td>SEPOL</td>
<td>170# 200 311 363 365 566 568 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 722</td>
</tr>
<tr>
<td>TEST1</td>
<td>691 697#</td>
</tr>
<tr>
<td>TEST2</td>
<td>688 711#</td>
</tr>
<tr>
<td>TEST3</td>
<td>685 722#</td>
</tr>
<tr>
<td>TIK</td>
<td>177# 250 451</td>
</tr>
<tr>
<td>TIINT</td>
<td>248#</td>
</tr>
<tr>
<td>TIRET</td>
<td>269#</td>
</tr>
<tr>
<td>TST11</td>
<td>699# 702</td>
</tr>
<tr>
<td>WDISP</td>
<td>552 606# 701</td>
</tr>
<tr>
<td>WDISP1</td>
<td>611 613#</td>
</tr>
</tbody>
</table>

**CROSS REFERENCE COMPLETE**
APPLICATION NOTE

January 1979

Serial I/O and Math Utilities for the 8049 Microcomputer

© Intel Corporation 1979

Lionel Smith and Cecil Moore
Microcomputer Applications
INTRODUCTION

The Intel® MCS-48 family of microcomputers marked the first time an eight bit computer with program storage, data storage, and I/O facilities was available on a single LSI chip. The performance of the initial processors in the family (the 8748 and the 8048) has been shown to meet or exceed the requirements of most current applications of microcomputers. A new member of the family, however, has been recently introduced which promises to allow the use of the single chip microcomputer in many application areas which have previously required a multichip solution. The Intel® 8049 virtually doubles processing power available to the systems designer. Program storage has been increased from 1K bytes to 2K bytes, data storage has been increased from 64 bytes to 128 bytes, and processing speed has been increased by over 80%. (The 2.5 microsecond instruction cycle of the first members of the family has been reduced to 1.36 microseconds.)

It is obvious that this increase in performance is going to result in far more ambitious programs being written for execution in a single chip microcomputer. This article will show how several program modules can be designed using the 8049. These modules were chosen to illustrate the capability of the 8049 in frequently encountered design situations. The modules included are full duplex serial I/O, binary multiply and divide routines, binary to BCD conversions, and BCD to binary conversion. It should be noted that since the 8049 is totally software compatible with the 8748 and 8048 these routines will also be useful directly on these processors. In addition the algorithms for these programs are expressed in a program design language format which should allow them to be easily understood and extended to suit individual applications with minimal problems.

FULL DUPLEX SERIAL COMMUNICATIONS

Serial communications have always been an important facet in the application of microprocessors. Although this has been partially due to the necessity of connecting a terminal to the microprocessor based system for program generation and debug, the main impetus has been the simple fact that a large share of microprocessors find their way into end products (such as intelligent terminals) which themselves depend on serial communication. When it is necessary to add a serial link to a microprocessor such as the Intel® MCS-85 or 86 the solution is easy; the Intel® 8251A USART or 8273 SDLC chip can easily be added to provide the necessary protocol. When it is necessary to do the same thing to a single chip microcomputer, however, the situation becomes more difficult.

Some microcomputers, such as the Intel 8048 and 8049 have a complete bus interface built into them which allows the simple connection of a USART to the processor chip. Most other single chip microprocessors, although lacking such a bus, can be connected to a USART with various artificial hardware and software constructs. The difficulty with using these chips, however, is more economic than technical; these same peripheral chips which are such a bargain when coupled to a microprocessor such as the MCS-85 or 86, have a significant cost impact on a single chip microcomputer based system. The high speed of the 8049, however, makes it feasible to implement a serial link under software control with no hardware requirements beyond two of the I/O pins already resident on the microcomputer.

There are many techniques for implementing serial I/O under software control. The application note “Application Techniques for the MCS-48 Family” describes several alternatives suitable for half duplex operation. Full duplex operation is more difficult, however, since it requires the receive and transmit processes to operate concurrently. This difficulty is made more severe if it is necessary for some other process to also operate while serial communication is occurring. Scanning a keyboard and display, for example, is a common operation of single chip microcomputer based system which might have to occur concurrently with the serial receive/transmit process. The next section will describe an algorithm which implements full duplex serial communication to occur concurrently with other tasks. The design goal was to allow 2400 baud, full duplex, serial communication while utilizing no more than 50% of the available processing power of the high speed 8049 microcomputer.

The format used for most asynchronous communication is shown in Figure 1. It consists of eight data bits with a leading 'START' bit and one or more trailing 'STOP' bits. The START bit is used to establish synchronization between the receiver and transmitter. The STOP bits ensure that the receiver will be ready to synchronize itself when the next start bit occurs. Two stop bits are normally used for 110 baud communication and one stop bit for higher rates.

![Figure 1](image)

The algorithm used for reception of the serial data is shown in Figure 2. It uses the on board timer of the 8049 to establish a sampling period of four times the desired baud rates. For 2400 baud operation a crystal frequency of 9,216 MHz was chosen after the following calculation:

\[ f = 480N(2400)(4) \]

where 480 is the factor by which the crystal frequency is divided within the processor to get the basic interrupt rate

2400 is the desired baud rate

4 is the required number of samples per bit time

N is the value loaded into the MCS-48 timer when it overflows.
The value N was chosen to be two (resulting in \( f = 9.216 \) MHz) so that the operating frequency of the 8049 could be as high as possible without exceeding the maximum frequency specification of the 8049 (11 MHz).

Once the meaning of these flags are understood the operation of the algorithm should be clear. The \textit{Receive Flag} is set whenever the program is in the process of receiving a character. The \textit{Synch Flag} is set when the center of the start bit has been checked and found to be a SPACE (if a MARK is detected at this point the receiver process has been triggered by a noise pulse so the program clears the \textit{Receive Flag} and returns to the idle state). When the program detects synchronization it loads the variable \textit{DATA} with 80H and starts sampling the serial line every four counts. As the data is received it is right shifted into variable \textit{DATA}; after eight bits have been received the initial one set into \textit{DATA} will result in a carry out and the program knows that it has received all eight bits. At this point it will transfer all eight bits to the variable \textit{OKDATA} and set the \textit{Byte Finished Flag} so that on the next sample it will test for a valid stop bit instead of shifting in data. If this test is successful the \textit{Data Ready Flag} will be set to indicate that the data is available to the main process. If the test is unsuccessful the \textit{Error Flag} will be set.

The transmit algorithm is shown in Figure 3. It is executed immediately following the receive process. It is a simple program which divides the free running clock down and transmits a bit every fourth clock. The variable \textit{TICK COUNTER} is used to do the division. The \textit{Transmitting Flag} indicates when a character transmission is in progress and is also used to determine when the START bit should be sent. The \textit{TICK COUNTER} is used to determine when to send the next bit (\textit{TICK COUNTER MOD 4 = 0}) and also when the STOP bits should be sent (\textit{TICK COUNTER = 9 4}). After the transmit routine completes any other timer based routines, such as a keyboard/display scanner or a real time clock, can be executed.

![Figure 2](image-url)

The timer interrupt service routine always loads the timer with a constant value. In effect the timer is used to generate an independent time base of four times the required baud rate. This time base is free running and is never modified by either the receive or transmit programs, thus allowing both of them to use the same timer. Routines which do other time dependent tasks (such as scanning keyboards) can also be called periodically at some fixed multiple of this basic time unit.

The algorithm shown in Figure 2 uses this basic clock plus a handful of flags to process the serial input data.
Figure 4 shows the complete receive and transmit programs as they are implemented in the instruction set of the 8049. Also included in Fig. 4 is a short routine which was used to test the algorithm.

ISIS-II MCS-48/UP-41 MACRO ASSEMBLER, V2.0

LOC OBJ SEG SOURCE STATEMENT

1:***************************************** *
2: * THIS PROGRAM TESTS THE FULL DUPLEX COMMUNICATION SOFTWARE *
3: * *
4: *
5:***************************************** *
6: 7 INCLUDE (.F1.UPTEST.PDL)
= 8: = 9: START OF TEST ROUTINE
= 10: = = = = = = = = = = = = = = = =
= 11: =
= 12: =
= 13: =
= 14: =
= 15: =
= 16: = ERROR COUNT =0
= 17: = REPEAT
= 18: = PATTERN =0
= 19: = INITIALIZE TIMER
= 20: = CLEAR FLAGBYTE
= 21: = FLAG1=0
= 22: = REPEAT
= 23: = IF TRANSMIT REQUEST FLAG =0 THEN
= 24: = NOP
= 25: = TRANSMIT REQUEST FLAG =1
= 26: = ENDIF
= 27: = IF DATA READY FLAG =1 THEN
= 28: = PATTERN =0
= 29: = DATA READY FLAG =0
= 30: = ENDIF
= 31: = UNTIL ERROR FLAG OR OVERRUN FLAG
= 32: = INCREMENT ERROR COUNT
= 33: = UNTIL FOREVER
= 34: = EOF
= 35: REJECT
= 36: 0000 ORG 0
= 37: 0000 C5 SELECT REGISTER BANK 0
= 38: 39: 1 GOTO TEST
= 40: 41: # INCLUDE (.F1.UART)
= 42: =
= 43: =
= 44: = ASYNCHRONOUS RECEIVE/TRANSMIT ROUTINE
= 45: = = = = = = = = = = = = = = = = = = = = =
= 46: = THIS ROUTINE RECEIVES SERIAL CODE USING PIN T0 AS RXD
= 47: = AND CONCURRENTLY TRANSMITS USING PIN P27
= 48: = NOTE
= 49: = THIS ROUTINE USES FLAG 1 TO BUFFER THE TRANSMITTED
Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>105</td>
<td>SOURCE STATEMENT</td>
</tr>
<tr>
<td>0000</td>
<td>106</td>
<td>= GENERAL CONSTANTS</td>
</tr>
<tr>
<td>0000</td>
<td>107</td>
<td>= 00H ; USED TO GENERATE A MARK</td>
</tr>
<tr>
<td>0000</td>
<td>108</td>
<td>= FF7F EQU NOT 00H ; USED TO GENERATE A SPACE</td>
</tr>
<tr>
<td>0000</td>
<td>109</td>
<td>= 0000 EQU 0 ; CONTROLS THE NUMBER OF STOP BITS</td>
</tr>
<tr>
<td>0000</td>
<td>110</td>
<td>= 111 EQU 1 ; 0 GENERATES ONE STOP BIT</td>
</tr>
<tr>
<td>0000</td>
<td>111</td>
<td>= 112 EQU 1 ; 1 GENERATES TWO STOP BITS</td>
</tr>
<tr>
<td>0000</td>
<td>113</td>
<td>= 114 ;</td>
</tr>
<tr>
<td>0000</td>
<td>115</td>
<td>= 115 EJECT</td>
</tr>
<tr>
<td>0000</td>
<td>116</td>
<td>= 117 ; START OF RECEIVE/TRANSMIT INTERRUPT SERVICE ROUTINE</td>
</tr>
<tr>
<td>0000</td>
<td>118</td>
<td>= 119 ;</td>
</tr>
<tr>
<td>0000</td>
<td>120</td>
<td>= 120 ; ORG 0007H</td>
</tr>
<tr>
<td>0000</td>
<td>121</td>
<td>= 122 ; 1 ENTER INTERRUPT MODE</td>
</tr>
<tr>
<td>0000</td>
<td>122</td>
<td>= 123 ; TISPBTS EQU 0 ; START OF RECEIVE ROUTINE</td>
</tr>
<tr>
<td>0000</td>
<td>123</td>
<td>= 124 ; RETP</td>
</tr>
<tr>
<td>0000</td>
<td>124</td>
<td>= 125 ; UART: SEL RB1</td>
</tr>
<tr>
<td>0000</td>
<td>125</td>
<td>= 126 ; 1 SAVE ACCUMULATOR CONTENTS</td>
</tr>
<tr>
<td>0000</td>
<td>126</td>
<td>= 127 ; MOV ATMP, A</td>
</tr>
<tr>
<td>0000</td>
<td>127</td>
<td>= 128 ; 1 RELOAD TIMER</td>
</tr>
<tr>
<td>0000</td>
<td>128</td>
<td>= 129 ; MOV A, #TIMCNT1</td>
</tr>
<tr>
<td>0000</td>
<td>129</td>
<td>= 130 ; MOV T.A</td>
</tr>
<tr>
<td>0000</td>
<td>130</td>
<td>= 131 ;</td>
</tr>
<tr>
<td>0000</td>
<td>131</td>
<td>= 132 ; OUTPUT TXD BUFFER (F1) TO TXD I/O LINE (P27)</td>
</tr>
<tr>
<td>0000</td>
<td>132</td>
<td>= 133 ;</td>
</tr>
<tr>
<td>0000</td>
<td>133</td>
<td>= 134 ;</td>
</tr>
<tr>
<td>0000</td>
<td>134</td>
<td>= 135 ; JF1 OMARK</td>
</tr>
<tr>
<td>0000</td>
<td>135</td>
<td>= 136 ; OSBP T2SPACE</td>
</tr>
<tr>
<td>0000</td>
<td>136</td>
<td>= 137 ; JMP RCV000</td>
</tr>
<tr>
<td>0000</td>
<td>137</td>
<td>= 138 ; OMARK ORL P2, #MARK</td>
</tr>
<tr>
<td>0000</td>
<td>138</td>
<td>= 139 ;</td>
</tr>
<tr>
<td>0000</td>
<td>139</td>
<td>= 140 ; START OF RECEIVE ROUTINE</td>
</tr>
<tr>
<td>0000</td>
<td>140</td>
<td>= 141 ;</td>
</tr>
<tr>
<td>0000</td>
<td>141</td>
<td>= 142 ;</td>
</tr>
<tr>
<td>0000</td>
<td>142</td>
<td>= 143 ; IF RECEIVE FLAG=0 THEN</td>
</tr>
<tr>
<td>0000</td>
<td>143</td>
<td>= 144 ; RCV000 MOV A, FLGBYT</td>
</tr>
<tr>
<td>0000</td>
<td>144</td>
<td>= 145 ; JB0 RCV010</td>
</tr>
<tr>
<td>0000</td>
<td>145</td>
<td>= 146 ; IF SERIAL INPUT=SPACE THEN</td>
</tr>
<tr>
<td>0000</td>
<td>146</td>
<td>= 147 ; JT0 XMIT</td>
</tr>
<tr>
<td>0000</td>
<td>147</td>
<td>= 148 ; RECEIVE FLAG = 1</td>
</tr>
<tr>
<td>0000</td>
<td>148</td>
<td>= 149 ; MOV A, FLGBYT</td>
</tr>
<tr>
<td>0000</td>
<td>149</td>
<td>= 150 ; ORL A, #RCYFLG</td>
</tr>
<tr>
<td>0000</td>
<td>150</td>
<td>= 151 ; BYTE FINISHED FLAG = 0</td>
</tr>
<tr>
<td>0000</td>
<td>151</td>
<td>= 152 ; ANL A, #NOT BYNFL</td>
</tr>
<tr>
<td>0000</td>
<td>152</td>
<td>= 153 ; ENDIF</td>
</tr>
<tr>
<td>0000</td>
<td>153</td>
<td>= 154 ; MOV FLGBYT, A</td>
</tr>
<tr>
<td>0000</td>
<td>154</td>
<td>= 155 ; JMP XMIT</td>
</tr>
<tr>
<td>0000</td>
<td>155</td>
<td>= 156 ; ELSE SINCE RECEIVE FLAG=1 THEN</td>
</tr>
<tr>
<td>0000</td>
<td>156</td>
<td>= 157 ; IF SYNC FLAG=0 THEN</td>
</tr>
<tr>
<td>0000</td>
<td>157</td>
<td>= 158 ; RCV010 JB1 RCV000</td>
</tr>
<tr>
<td>0000</td>
<td>158</td>
<td>= 159 ; IF SERIAL INPUT=SPACE THEN</td>
</tr>
</tbody>
</table>

Figure 4 (continued)
```
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0026</td>
<td>3633</td>
<td>160</td>
<td>JTO RCV20</td>
</tr>
<tr>
<td>0026</td>
<td>4302</td>
<td>161</td>
<td>SYNC FLAG =1</td>
</tr>
<tr>
<td>0026</td>
<td>HE</td>
<td>162</td>
<td>ORL A #SYNCFLG</td>
</tr>
<tr>
<td>0026</td>
<td>B021</td>
<td>163</td>
<td>MOV FLSBYT.A</td>
</tr>
<tr>
<td>0026</td>
<td>B080</td>
<td>164</td>
<td>DATA =O0H</td>
</tr>
<tr>
<td>0026</td>
<td>B044</td>
<td>165</td>
<td>MOV B0,#DATA</td>
</tr>
<tr>
<td>0026</td>
<td></td>
<td>166</td>
<td>MOV AO #O0H</td>
</tr>
<tr>
<td>0026</td>
<td></td>
<td>167</td>
<td>SAMPLE CNTR =4</td>
</tr>
<tr>
<td>0026</td>
<td>B044</td>
<td>168</td>
<td>MOV SAMCTR #4</td>
</tr>
<tr>
<td>0026</td>
<td>0464</td>
<td>169</td>
<td>JMP XMIT</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>170</td>
<td>ELSE SINCE SERIAL INPUT=MARK THEN</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>171</td>
<td>RECEIVE FLAG =0</td>
</tr>
<tr>
<td>0027</td>
<td>33FE</td>
<td>172</td>
<td>RCV020 ANL A #NOT RCVFLG</td>
</tr>
<tr>
<td>0027</td>
<td>AE</td>
<td>173</td>
<td>ENDF</td>
</tr>
<tr>
<td>0027</td>
<td>0464</td>
<td>174</td>
<td>MOV FLSBYT.A</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>175</td>
<td>JMP XMIT</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>176</td>
<td>ELSE SINCE SYNC FLAG=1 THEN</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>177</td>
<td>SAMPLE COUNTER =SAMPLE COUNTER+1</td>
</tr>
<tr>
<td>0027</td>
<td>ED64</td>
<td>178</td>
<td>RCV030 DJNZ SAMCTR XMIT</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>179</td>
<td>IF SAMPLE COUNTER=0 THEN</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>180</td>
<td>SAMPLE COUNTER =4</td>
</tr>
<tr>
<td>0027</td>
<td>B044</td>
<td>181</td>
<td>MOV SAMCTR #4</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>182</td>
<td>IF BYTE FINISHED FLAG=0 THEN</td>
</tr>
<tr>
<td>0027</td>
<td>5259</td>
<td>183</td>
<td>JBC RCV50</td>
</tr>
<tr>
<td>0027</td>
<td>97</td>
<td>184</td>
<td>CLR C</td>
</tr>
<tr>
<td>0027</td>
<td>2642</td>
<td>185</td>
<td>CARRY =SERIAL INPUT</td>
</tr>
<tr>
<td>0027</td>
<td>H7</td>
<td>186</td>
<td>JNTO RCV040</td>
</tr>
<tr>
<td>0027</td>
<td>B021</td>
<td>187</td>
<td>CPL C</td>
</tr>
<tr>
<td>0027</td>
<td>F0</td>
<td>188</td>
<td>RCV040 MOV R0,#DATA</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>189</td>
<td>MOV A, #00</td>
</tr>
<tr>
<td>0027</td>
<td>67</td>
<td>190</td>
<td>JMP XMIT</td>
</tr>
<tr>
<td>0027</td>
<td>R0</td>
<td>191</td>
<td>RRC A</td>
</tr>
<tr>
<td>0027</td>
<td>46</td>
<td>192</td>
<td>MOV A, #00</td>
</tr>
<tr>
<td>0027</td>
<td>664</td>
<td>193</td>
<td>IF CARRY=1 THEN</td>
</tr>
<tr>
<td>0027</td>
<td>E64</td>
<td>194</td>
<td>XMIT</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>195</td>
<td>JK DATA =DATA</td>
</tr>
<tr>
<td>0027</td>
<td>B620</td>
<td>196</td>
<td>MOV R0,#00DAT</td>
</tr>
<tr>
<td>0027</td>
<td>R0</td>
<td>197</td>
<td>MOV A, #00</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>198</td>
<td>IF DATA READY FLAG=0 THEN</td>
</tr>
<tr>
<td>0027</td>
<td>FE</td>
<td>199</td>
<td>MOV A, FLSBYT</td>
</tr>
<tr>
<td>0027</td>
<td>7254</td>
<td>200</td>
<td>JB3 RCV045</td>
</tr>
<tr>
<td>0027</td>
<td>4394</td>
<td>201</td>
<td>BYTE FINISHED FLAG = 1</td>
</tr>
<tr>
<td>0027</td>
<td>4394</td>
<td>202</td>
<td>ORL A #BYNFL</td>
</tr>
<tr>
<td>0027</td>
<td>AE</td>
<td>203</td>
<td>MOV FLSBYT.A</td>
</tr>
<tr>
<td>0027</td>
<td>0464</td>
<td>204</td>
<td>JMP XMIT</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>205</td>
<td>ELSE</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>206</td>
<td>BYTE FINISHED FLAG = 1</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>207</td>
<td>OVERRUN FLAG = 1</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>208</td>
<td>RCV045</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>209</td>
<td>MOV A, FLSBYT</td>
</tr>
<tr>
<td>0027</td>
<td>4394</td>
<td>210</td>
<td>ORL A #BYNFL OR OVRUN</td>
</tr>
<tr>
<td>0027</td>
<td>RE</td>
<td>211</td>
<td>MOV FLSBYT.A</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>212</td>
<td>ENDF</td>
</tr>
<tr>
<td>0027</td>
<td></td>
<td>213</td>
<td>ENDIF</td>
</tr>
<tr>
<td>0027</td>
<td>0464</td>
<td>214</td>
<td>JMP XMIT</td>
</tr>
</tbody>
</table>

Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

1-56
LOC OBJ  SEQ  SOURCE STATEMENT

= 215 : 4  ELSE  SINCE BYTE FINISHED FLAG=1 THEN
= 216 : 5  IF SERIAL INPUT=MARK THEN

0059 265F  = 217 RCV050: INTO  RCV060
0059 4308  = 218 : 6  DATA READY FLAG=1
005A 0461  = 219  OPL  A. NOTDYFL
005B 0461  = 220  JMP  RCV070
005C 0461  = 221 : 5  ELSE  SINCE SERIAL INPUT=SPACE THEN
005D 53FC  = 222 RCV060: ORL  A. REPLFLG
0061 53FC  = 223 : 5  ENDIF
0062 53FC  = 224 : 5  RECEIVE FLAG=0
0063 53FC  = 225 : 5  SYNC FLAG=0
0064 53FC  = 226 : 5  IF SERIAL INPUT=MARK
0065 53FC  = 227 RCV070: ANL  A. NOT(SYNFLG OR RCVFLG)
0066 53FC  = 228  MOV  FLGBYT.A
0067 53FC  = 229 : 4  ENDIF
0068 53FC  = 230 : 2  ENDIF
0069 53FC  = 231 : 2  ENDIF
006A 53FC  = 232 : 1  ENDIF
006B 53FC  = 233  REJECT
= 234 ;
= 235 ;  START OF TRANSMIT ROUTINE
= 236 ;  ========================
= 237 ;
= 238 : 1
= 239 : TRANSMITTER OUTPUT BIT IS P2-7
= 240 : 1 TCK COUNTER = TCK COUNTER+1

0064 1C  = 241 XMIT:  INC  TCKCTR
= 242 : 1 IF TCK COUNTER MOD 4=0 THEN

0065 2303  = 243  MOV  A. #03H
0065 9697  = 244  ANL  A. TCKCTR
0066 53FC  = 245  JNZ  RETURN
= 246 : 2  IF TRANSMITTING FLAG=1 THEN

0067 53FC  = 247  MOV  A. FLGBYT
0068 53FC  = 248  CPL  A
0069 53FC  = 249  JB6  XMT040
006A 53FC  = 250  IF STPBTS EQ 1
= 251 : 3  IF TCK COUNTER=00 1010 00 BINARY THEN
= 252  MOV  A. #28H ; CONDITIONAL ASSEMBLY
= 253  XRL  A. TCKCTR
= 254  JNZ  XMT010
= 255 : 4  TRANSMITTING FLAG:=0
= 256  MOV  A. FLGBYT
= 257  ANL  A. NOT TRANFL
= 258  MOV  FLGBYT.R
= 259  JMP  RETURN
= 260  ENDIF
= 261 : 3  ELSE  IF TCK COUNTER=00 1001 00 BINARY THEN

006E 2224  = 262 XMT010:  MOV  A. #24H
006F 2224  = 263  XRL  A. TCKCTR
0070 2224  = 264  JNZ  XMT020
= 265 : 4  SEND END MARK

0071 2224  = 266  CLR  F1 ; SET FLAG1 TO MARK
0072 2224  = 267  CPL  F1
0073 2224  = 268  IF STPBTS EQ 0
= 269 : 4  TRANSMITTING FLAG:=0

Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

1-57
Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

1-58
LUC OBJ SEQ SOURCE STATEMENT

325 ;
326 ;
327 ; 1 ERROR COUNT = 0
0100 BF00 328 TEST: MOV ERRCNT, 0
329 ; 1 REPEAT
330 TLOP
331 ; 2 PATTERN = 0
0102 BE00 332 MOV PATT, #00
333 ; 2 INITIALIZE TIMER
0104 23FE 334 MOV A, #TINTCNT
0106 62 335 MOV T, A
0107 55 336 STR T
0108 25 337 EN TONI
338 ; 2 CLEAR FLAGBYTE
0109 B81E 339 MOV R0, #FLAGBYTE
010A B800 340 MOV @R0, #0
341 ; 2 FLAG = MARK
010D A5 342 CLR F1
010E B5 343 CPL F1
344 ; 2 REPEAT
345 TLOP:
346 ; 3 IF TRANSMIT REQUEST FLAG = 0 THEN
010F B81E 347 MOV R0, #FLAGBYTE
0111 F0 348 MOV A, #0
0112 B224 349 JB5 TREC
350 ; 4 NOTBYTE = PATTERN
0114 B923 351 MOV R1, #NOTBYTE
0116 FE 352 MOV A, PATT
0117 A1 353 MOV @R1, A
354 ; 4 TRANSMIT REQUEST FLAG = 1
0118 35 355 DIS TONI ; LOCK OUT TIMER INTERRUPT
356 ; 3 IF DATA READY FLAG = 1 THEN
0119 F0 357 ; SO THAT MUTUAL EXCLUSION IS MAINTAINED WHILE
011A 4320 358 MOV A, #00
011C A0 359 ORL A, #TROFL
011D 25 360 MOV @R0, A
011E 1622 361 EN TONI
0120 2424 362 JTF TESTA
0122 140A 363 JMP TREC
0124 35 364 TESTA: CALL UART ; CALL UART BECAUSE TIMER OVERFLOWED DURING LOCKOUT
365 ; 3 ENDIF
366 ; 3 IF DATA READY FLAG = 1 THEN
0126 7238 367 TREC:
0127 F0 368 MOV A, #00
012B 37 369 CPL A
012C 7228 370 JB3 TREC
371 ; 4 PATTERN = OKDATA
012D 8920 372 MOV R1, #OKDAT
012F F1 373 MOV A, #R1
012B AE 374 MOV PATT, A
375 ; 4 DATA READY FLAG = 0
012C 35 376 DIS TONI ; LOCK OUT TIMER INTERRUPT
377 ; 3 IF DATA READY FLAG = 1 THEN
012D F0 378 ; SO THAT MUTUAL EXCLUSION IS MAINTAINED WHILE
379 MOV A, #00

Figure 4 (continued)
LOC OBJ SEQ SOURCE STATEMENT

012E 53F7 380 ANL A, #NOT DRDYFL
0130 90 381 MOV @R0, A
0131 25 382 EN TENTI
0132 1636 383 JTF TESTB
0134 2438 384 JMP TRECE
0136 140A 385 TESTB: CALL UART ; CALL UART IF TIMER OVERFLOWED DURING LOCKOUT
386 TRECE:
387 J3 ENIF
388 J2 UNTIL ERROR FLAG OR OVERRUN FLAG
0138 F0 389 MOV A, @R0
0139 5390 390 ANL A, #(OVERUN OR ERRFLG)
013C 08F 391 JZ TLOP
392 J2 INCREMENT ERROR COUNT
013D 1F 393 INC ERRCNT
394 J1 UNTIL FOREVER
013E 2402 395 JMP TLOP
396 E0F UNTIL
397 END

USER SYMBOLS
ATEMP 0007 EVENFL 0004 DROYFL 0008 ERRCNT 0007 ERRFLG 0010 FLGNOT 0006 MARK 0080 MDATA 0021
MFLAGV 001E MINTBV 0023 MOKDAT 0020 NSAMCT 0010 MTCTCT 001C MINTBY 0022 ORMARK 0015 OSPEC 0011
OVEV 0090 PATT 0096 RCYR00 0017 KVC010 0024 RCYB02 0033 RCYB09 0038 RCYB04 0042 KVC004 0054
RCYV05 0059 RCYB10 0065 RCYV07 0061 RCYV01 0017 RCYV00 0010 RCYV09 0038 RCYV04 0042 KVC004 0054
STPMTS 0000 UNFLG 0002 TCKCTR 0004 TEST 0100 TESTB 0122 TESRD 0136 TLOP 018F TSTCN 0FFE
T1SR 0007 TLOP 0182 TREC 0124 TRECE 0138 TRNGFL 0040 TRGFL 0028 UART 000A XMIT 0064
XINTO10 006E XINTO20 007B XINTO40 0095

ASSEMBLY COMPLETE: NO ERRORS

Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

MULTIPLY ALGORITHMS

Most microcomputer programmers have at one time or another implemented a multiply routine as part of a larger program. The usual procedure is to find an algorithm that works and modify it to work on the machine being used. There is nothing wrong with this approach. If engineers felt that they had to reinvent the wheel every time a new design is undertaken, that's probably what most of us would be doing—designing wheels. If the efficiency of the multiply algorithm, either in terms of code size or execution time is important, however, it is necessary to be reasonably familiar with the multiplication process so that appropriate optimizations for the machine being used can be made.

To understand how multiplication operates in the binary number system, consider the multiplication of two four bit operands A and B. The “ones and zeros” in A and B represent the coefficients of two polynomials. The operation A × B can be represented as the following multiplication of polynomials:

\[
A3\cdot2^3 + A2\cdot2^2 + A1\cdot2^1 + A0\cdot2^0
\]

\[
X \quad B3\cdot2^3 + B2\cdot2^2 + B1\cdot2^1 + B0\cdot2^0
\]

\[
+ B0A3\cdot2^3 + B0A2\cdot2^2 + B0A1\cdot2^1 + B0A0\cdot2^0
\]

\[
+ B1A3\cdot2^4 + B1A2\cdot2^3 + B1A1\cdot2^2 + B1A0\cdot2^1
\]

\[
+ B2A3\cdot2^5 + B2A2\cdot2^4 + B2A1\cdot2^3 + B2A0\cdot2^2
\]

\[
+ B3A3\cdot2^6 + B3A2\cdot2^5 + B3A1\cdot2^4 + B3A0\cdot2^3
\]

1-60
The sum of all these terms represents the product of \( A \) and \( B \). The simplest multiply algorithm factors the above terms as follows:

\[
A^*B = B_0^*(A)^*2^0 + B_1^*(A)^*2^1 + B_2^*(A)^*2^2 + B_3^*(A)^*2^3
\]

Since the coefficients of \( B \) (i.e., \( B_0, B_1, B_2, \) and \( B_3 \)) can only take on the binary values of 1 or 0, the sum of the products can be formed by a series of simple adds and multiplications by two. The simplest implementation of this would be:

MULTIPLY:

PRODUCT := 0
IF \( B_0 = 1 \) THEN PRODUCT := PRODUCT + A
IF \( B_1 = 1 \) THEN PRODUCT := PRODUCT + 2*A
IF \( B_2 = 1 \) THEN PRODUCT := PRODUCT + 4*A
IF \( B_3 = 1 \) THEN PRODUCT := PRODUCT + 8*A
END MULTIPLY

In order to conserve memory, the above straight line code is normally converted to the following loop:

MULTIPLY:

PRODUCT := 0
COUNT := 4
REPEAT
IF \( B[0] = 1 \) THEN PRODUCT := PRODUCT + A
A := 2*A
B := B/2
COUNT := COUNT - 1
UNTIL COUNT := 0
END MULTIPLY

The repeated multiplication of \( A \) by two (which can be performed by a simple left shift) forms the terms \( 2*A, 4*A, \) and \( 8*A \). The variable \( B \) is divided by two (performed by a simple right shift) so that the least significant bit can always be used to determine whether the addition should be executed during each pass through the loop. It is from these shifting and addition operations that the “shift and add” algorithm takes its common name.

The “shift and add” algorithm shown above has two areas where efficiency will be lost if implemented in the manner shown. The first problem is that the addition to the partial product is double precision relative to the two operands. The other problem, which is also related to double precision operations, is that the \( A \) operand is double precision and that it must be left shifted and then the \( B \) operand must be right shifted. An examination of the “longhand” polynomial multiplication will reveal that, although the partial product is indeed double precision, each addition performed is only single precision. It would be desirable to be able to shift the partial product as it is formed so that only single precision additions are performed. This would be especially true if the partial product could be shifted into the “\( B \)” operand since one bit of the partial product is formed during each pass through the loop and (happily) one bit of the “\( B \)” operand is vacated. To do this, however, it is necessary to modify the algorithm so that both of the shifts that occur are of the same type.

To see how this can be done one can take the basic multiplication equation already presented:

\[
A^*B = 2^4[B_0^*(A)^*2^0 + B_1^*(A)^*2^1 + B_2^*(A)^*2^2 + B_3^*(A)^*2^3]
\]

and factoring \( 2^4 \) from the right side:

\[
A^*B = 2^4[B_0^*(A^*2^{-4}) + B_1^*(A^*2^{-3}) + B_2^*(A^*2^{-2}) + B_3^*(A^*2^{-1})]
\]

This operation has resulted in a term (within the brackets) which can be formed by right shifts and adds and then multiplied by \( 2^4 \) to get the final result. The resulting algorithm, expanded to form an eight by eight multiplication, is shown in figure 5. Note that although the result is a full sixteen bits, the algorithm only performs eight bit additions and that only a single sixteen bit shift operation is involved. This has the effect of reducing both the code space and the execution time for the routine.

**Figure 5**

All mnemonics copyrighted © Intel Corporation 1979.
$include(f1:mpy8 pdl)

= 24 :// mpysx8.
= 25 :// MULTIPICAND[15-8] = 0
= 26 :// COUNT = 0
= 27 :// REPEAT
= 28 :// IF MULTIPICAND[0] = 0 THEN BEGIN
= 29 :// MULTIPICAND := MULTIPICAND / 2
= 30 :// ELSE
= 31 :// MULTIPICAND := MULTIPICAND + MULTIPLIER
= 32 :// ENDIF
= 33 :// COUNT := COUNT - 1
= 34 :// UNTIL COUNT = 0
= 35 :// END MPYSOS

= 36 :// MOTO S
= 37 :// EQUATES
= 38 :// =====
= 39 ://
= 40 :

0002 41 XA EQU $2
= 41 ://
= 42 COUNT EQU 8
= 43 CNT EQU 4
= 44 :

0003 45 DIGIT EQU 3
= 46 :
= 47 $EJECT
= 48 #include(f1:mpy8)
= 49 :// mpysx8
= 50 mpysx8:
= 51 ;// MULTIPICAND[15-8] = 0
= 52 MOV XA, #0
= 53 ;// COUNT := 0
= 54 MOV COUNT, #0
= 55 ;// REPEAT
= 56 MPY8LP
= 57 ;// IF MULTIPICAND[0] = 0 THEN BEGIN
= 58 JBB MPY8A
= 59 ;// MULTIPICAND := MULTIPICAND / 2
= 60 XCH A, XA
= 61 CLR C
= 62 RRC A
= 63 XCH A, XA
= 64 RRC A
= 65 D.NZ COUNT MPY8LP
= 66 RET
= 67 ;// ELSE

Figure 5 (continued)

All mnemonics copyrighted © Intel Corporation 1979.
DIVIDE ALGORITHMS

In order to understand binary division a four bit operation will again be used as an example. The following algorithm will perform a four by four division:

DIVIDE:
  IF 16*DIVISOR>= DIVIDEND THEN
    SET OVERFLOW ERROR FLAG
  ELSE
    IF 8*DIVISOR>= DIVIDEND THEN
      QUOTIENT[3]:= 1
      DIVIDEND:= DIVIDEND- 8*DIVISOR
    ELSE
      QUOTIENT[3]:= 0
    ENDIF
    IF 4*DIVISOR>= DIVIDEND THEN
      QUOTIENT[2]:= 1
      DIVIDEND:= DIVIDEND- 4*DIVISOR
    ELSE
      QUOTIENT[2]:= 0
    ENDIF
    IF 2*DIVISOR>= DIVIDEND THEN
      QUOTIENT[1]:= 1
      DIVIDEND:= DIVIDEND- 2*DIVISOR
    ELSE
      QUOTIENT[1]:= 0
    ENDIF
    IF 1*DIVISOR>= DIVIDEND THEN
      QUOTIENT[0]:= 1
      DIVIDEND:= DIVIDEND- 1*DIVISOR
    ELSE
      QUOTIENT[0]:= 0
    ENDIF
  ENDIF
END

The algorithm is easy to understand. The first test asks if the division will fit into the dividend sixteen times. If it will, the quotient cannot be expressed in only four bits so an overflow error flag is set and the divide algorithm ends. The algorithm then proceeds to determine if eight times the divisor fits, four times, etc. After each test it either sets or clears the appropriate quotient bit and modifies the dividend. To see this algorithm in action, consider the division of 15 by 5:

---

The result is Q=0011 which is the binary equivalent of 3—the correct answer. Clearly this algorithm can (and has been) converted to a loop and used to perform divisions. An examination of the procedure, however, will show that it has the same problems as the original multiply algorithm.
The first problem is that double precision operations are involved with both the comparison of the division with the dividend and the conditional subtraction. The second problem is that as the quotient bits are derived they must be shifted into a register. In order to reduce the register requirements, it would be desirable to shift them into the divisor register as they are generated since the divisor register gets shifted anyway. Unfortunately the quotient bits are derived most significant bits first so doing this will form a mirror image of the quotient—not very useful.

Both of these problems can be solved by observing that the algorithm presented for divide will still work if both sides of all the "equations" involving the dividend are divided by sixteen. The looping algorithm then would proceed as follows:

**DIVIDE:**

```plaintext
QUOTIENT := 0
COUNT := 4
DIVIDEND := DIVIDEND/16
IF DIVISOR > DIVIDEND THEN
    OVERFLOW FLAG := 1
ELSE
    REPEAT
        DIVIDEND := DIVIDEND*2
        QUOTIENT := QUOTIENT*2
        IF DIVISOR > DIVIDEND THEN
            QUOTIENT := QUOTIENT + 1/*SET QUOTIENT[0]*/
            DIVIDEND := DIVIDEND - DIVISOR
        ENDIF
        COUNT := COUNT - 1
    UNTIL COUNT = 0
ENDIF
END DIVIDE
```

When this algorithm is implemented on a computer which does not have a direct compare instruction the comparison is done by subtraction and the inner loop of the algorithm is modified as follows:

```
REPEAT
    DIVIDEND := DIVIDEND*2
    QUOTIENT := QUOTIENT*2
    IF DIVISOR > DIVIDEND THEN
        QUOTIENT := QUOTIENT + 1
    ELSE
        DIVIDEND := DIVIDEND + DIVISOR
    ENDIF
    COUNT := COUNT - 1
UNTIL COUNT = 0
```

An implementation of this algorithm using the 8049 instruction set is shown in figure 6. This routine does an unsigned divide of a 16 bit quantity by an eight bit quantity. Since the multiply algorithm of figure 5 generates a 16 bit result from the multiplication of two eight bit operands, these two routines complement each other and can be used as part of more complex computations.
LOC OBJ | SEQ | SOURCE STATEMENT

= 18 :+ | C = SET IF OVERFLOW ELSE CLEARED
= 19 :+ *
= 20 :+ **************************************************************************
21 :
22 :
23 :!INCLUDE( F1.DIV16 PD1)
24 :!DIV16
25 :! COUNT = 8
26 :! DIVIDEND(15-8)=DIVIDEND(15-8)\DIVISOR
27 :! IF BORROW=0 THEN /* IT FITS*/
28 :! SET OVERFLOW FLAG
29 :! ELSE
30 :! RESTORE DIVIDEND
31 :! REPEAT
32 :! DIVIDEND=DIVIDEND\2
33 :! QUOTIENT=QUOTIENT\2
34 :! DIVIDEND(15-8)=DIVIDEND(15-8)\DIVISOR
35 :! IF BORROW=1 THEN
36 :! RESTORE DIVIDEND
37 :! ELSE
38 :! QUOTIENT[0] = 1
39 :! ENDIF
40 :! COUNT = COUNT\-1
41 :! UNTIL COUNT=0
42 :! CLEAR OVERFLOW FLAG
43 :! ENDIF
44 :! ENDDIVIDE
45 :
46 :! EQUATES
47 :==========
48 :
0002 = 49 \X9 EQU R2
0003 = 50 COUNT EQU R3
51 :
52 \REJECT
53 :! INCLUDE( F1.DIV16)
= 54 :! DIV16:
0000 2A = 55 DIV16 ;XCH A, XA ; ROUTINE WORKS MOSTLY WITH B11S 15-8
= 56 :! COUNT = 8
0001 8B08 = 57 MOV COUNT, \H:
= 58 :! DIVIDEND(15-8)\DIVIDEND(15-8)\DIVISOR
0002 37 = 59 CPL A
0004 61 = 60 ADD A, \R1
0005 37 = 61 CPL A
0006 F608 = 62 ;! IF BORROW=0 THEN /* IT FITS*/
0006 FE = 63 JC DIV16
0006 H7 = 64 ;! SET OVERFLOW FLAG
0009 0424 = 65 CPL C
000A 057F = 66 JMP DIV1B
000B 7F = 67 ;! ELSE
000C 80 DIV16:
= 69 ;! RESTORE DIVIDEND
000D 61 = 70 ADD A, \R1
000F 2F = 71 ;! REPEAT
0011 2F = 72 DIV16:
= 72 ;! DIVIDEND=DIVIDEND\2

Figure 6 (continued)
BINARY AND BCD CONVERSIONS

The conversion of a binary value to a BCD (binary coded decimal) number can be done with a very straightforward algorithm:

```
CONVERT_TO_BCD:
  BCDACCUM = 0
  COUNT = PRECISION
  REPEAT
    BIN: = BIN * 2
    BCD: = BCD * 2 + CARRY
    COUNT: = COUNT - 1
  UNTIL COUNT = 0
  END CONVERT_TO_BCD
```

The variable `BCDACCUM` is a BCD string used to accumulate the result; the variable `BIN` is the binary number to be converted. `PRECISION` is a constant which gives the length, in binary bits of `BIN`. To see how this works, assume that `BIN` is a sixteen bit value with the most significant bit set. On the first pass through the loop the multiplication of `BIN` will result in a carry and this carry will be added to `BCD`. On the remaining passes through the loop BCD will be multiplied by two 15 times. The initial carry into BCD will be multiplied by \(2^{15}\) or 32768, which is the "value" of the most significant bit of `BIN`. The process repeats with each bit of `BIN` being introduced to `BCDACCUM` and then being scaled up on successive passes through the loop. Figure 7 shows the implementation of this algorithm for the 8049.

All mnemonics copyrighted © Intel Corporation 1979.

Figure 6 (continued)
LOC OBJ  SEQ  SOURCE STATEMENT

1 #MACROFILE
2 #INCLUDE F1(CONBCD.MAC)
3 ;**********************************************************************
4 ;
5 ; CONBCD
6 ;
7 ;**********************************************************************
8 ;
9 ; THIS UTILITY CONVERTS A 16 BIT BINARY VALUE TO BCD
10 ;
11 ; AT ENTRY
12 ;
13 ; A = LOWER EIGHT BITS OF BINARY VALUE
14 ;
15 ; R8 = POINTER TO FACED BCD STRING
16 ;
17 ; AT EXIT
18 ;
19 ; C = SET IF OVERFLOW ELSE CLEARED
20 ;
21 ;
22 #INCLUDE F1(CONBCD.PDL)
23 ; CONVTO BCD
24 ;
25 ; BCDACC = 0
26 ; COUNT = 16
27 ; REPEAT
28 ;
29 ; BIN = BIN + 2
30 ; BCD = BCD + 2 + CARRY
31 ; IF CARRY FROM BCDACC GOTO ERROR EXIT
32 ; COUNT = COUNT - 1
33 ; UNTIL COUNT = 0
34 ;
35 ; END CONVTO BCD
36 ;
37 ; EQUATES
38 ;
39 ; COUNT= EQU R3
40 ; ICNT= EQU R4
41 ;
42 ; DIGFR= EQU 3
43 ;
44 ; REJECT
45 ;
46 ; INCLUDE F1(CONBCD)
47 ;
48 ;
49 ; CONVTO BCD
50 ;
51 ; BCDACC = 0
52 ;
53 ; XCH A R8

Figure 7

All mnemonics copyrighted © Intel Corporation 1979.

1-67
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SE0</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>28</td>
<td>= 54</td>
<td>XCH A, R0</td>
</tr>
<tr>
<td>0003</td>
<td>00</td>
<td>= 55</td>
<td>MOV INT. #DIGR</td>
</tr>
<tr>
<td>0004</td>
<td>8100</td>
<td>= 56</td>
<td>BCDCCO MOV @#1.000</td>
</tr>
<tr>
<td>0005</td>
<td>19</td>
<td>= 57</td>
<td>INC F1</td>
</tr>
<tr>
<td>0006</td>
<td>ECD5</td>
<td>= 58</td>
<td>DJNZ INT-BDCDDO</td>
</tr>
<tr>
<td>0007</td>
<td>8810</td>
<td>= 59</td>
<td>:1 COUNT =1;</td>
</tr>
<tr>
<td>0008</td>
<td>8810</td>
<td>= 60</td>
<td>MOV COUNT =16</td>
</tr>
<tr>
<td>0009</td>
<td>8810</td>
<td>= 61</td>
<td>:1 REPEAT</td>
</tr>
<tr>
<td>0010</td>
<td>8810</td>
<td>= 62</td>
<td>BCDCCOB</td>
</tr>
<tr>
<td>0011</td>
<td>97</td>
<td>= 63</td>
<td>EIN =BIN#2</td>
</tr>
<tr>
<td>0012</td>
<td>F7</td>
<td>= 64</td>
<td>CLR C</td>
</tr>
<tr>
<td>0013</td>
<td>2A</td>
<td>= 65</td>
<td>RLC A</td>
</tr>
<tr>
<td>0014</td>
<td>2A</td>
<td>= 66</td>
<td>XCH A, VA</td>
</tr>
<tr>
<td>0015</td>
<td>F7</td>
<td>= 67</td>
<td>RLC A</td>
</tr>
<tr>
<td>0016</td>
<td>2A</td>
<td>= 68</td>
<td>XCH A, VA</td>
</tr>
<tr>
<td>0017</td>
<td>28</td>
<td>= 69</td>
<td>EBCD =BCD+2+CARRY</td>
</tr>
<tr>
<td>0018</td>
<td>29</td>
<td>= 70</td>
<td>XCH A, R0</td>
</tr>
<tr>
<td>0019</td>
<td>29</td>
<td>= 71</td>
<td>MOV R1, A</td>
</tr>
<tr>
<td>001A</td>
<td>28</td>
<td>= 72</td>
<td>XCH A, R0</td>
</tr>
<tr>
<td>001B</td>
<td>BC03</td>
<td>= 73</td>
<td>MOV INT. #DIGR</td>
</tr>
<tr>
<td>001C</td>
<td>00</td>
<td>= 74</td>
<td>MOV TEMP1, A</td>
</tr>
<tr>
<td>001D</td>
<td>FI</td>
<td>= 75</td>
<td>BCDCC MOV A, R0</td>
</tr>
<tr>
<td>001E</td>
<td>71</td>
<td>= 76</td>
<td>ADDC A, R0</td>
</tr>
<tr>
<td>001F</td>
<td>57</td>
<td>= 77</td>
<td>DA A</td>
</tr>
<tr>
<td>0020</td>
<td>A1</td>
<td>= 78</td>
<td>MOV @R1, A</td>
</tr>
<tr>
<td>0021</td>
<td>19</td>
<td>= 79</td>
<td>INC R1</td>
</tr>
<tr>
<td>0022</td>
<td>EC17</td>
<td>= 80</td>
<td>DJNZ INT-BCDDC</td>
</tr>
<tr>
<td>0023</td>
<td>FD</td>
<td>= 81</td>
<td>MOV A, TEMP1</td>
</tr>
<tr>
<td>0024</td>
<td>F624</td>
<td>= 82</td>
<td>:2 IF CARRY FROM BCDACC GOTO ERROR EXIT</td>
</tr>
<tr>
<td>0025</td>
<td>3</td>
<td>= 83</td>
<td>JC BCDCC</td>
</tr>
<tr>
<td>0026</td>
<td>84</td>
<td>= 84</td>
<td>:2 COUNT =COUNT-1</td>
</tr>
<tr>
<td>0027</td>
<td>00</td>
<td>= 85</td>
<td>UNTIL COUNT=0</td>
</tr>
<tr>
<td>0028</td>
<td>EC17</td>
<td>= 86</td>
<td>DJNZ COUNT=BCDDC</td>
</tr>
<tr>
<td>0029</td>
<td>97</td>
<td>= 87</td>
<td>CLR C</td>
</tr>
<tr>
<td>002A</td>
<td>83</td>
<td>= 88</td>
<td>END CONVERT TO BCD</td>
</tr>
<tr>
<td>002B</td>
<td>89</td>
<td>= 89</td>
<td>BCDCC RD</td>
</tr>
<tr>
<td>002C</td>
<td>90</td>
<td>= 90</td>
<td>END</td>
</tr>
</tbody>
</table>

**USER SYMBOLS**

| BCDCC | 0005 | BCDCCO 000C | BCDCCO 0024 | BCDCCO 0017 | CNBCC 0000 | COUNT 0003 | DIGPR 0003 | INT 0004 |
| TEMP1 | 0005 | VA | 0002 |

**ASSEMBLY COMPLETE. NO ERRORS**

Figure 7 (continued)

All mnemonics copyrighted © Intel Corporation 1979.
The conversion of a BCD value to binary is essentially the same process as converting a binary value to BCD. The only complexity is the two multiplications by ten. The BCDACCUM can be multiplied by ten by shifting it left four places (one digit). The variable BIN could be multiplied using the multiply algorithm already discussed, but it is usually more efficient to do this by making the following substitution:

\[
\text{BIN} = 10 \cdot \text{BIN} = (2) \cdot (5) \cdot (\text{BIN}) = 2 \cdot (2 \cdot 2 + 1) \cdot \text{BIN}
\]

This implies that the value \(10 \cdot \text{BIN}\) can be generated by saving the value of \text{BIN} and then shifting \text{BIN} two places left. After this the original value of \text{BIN} can be added to the new value of \text{BIN} (forming \(5 \cdot \text{BIN}\)) and then \text{BIN} can be multiplied by two. It is often possible to implement the multiplication of a value by a constant by using such techniques. Figure 8 shows an 8049 routine which converts BCD values to binary. This routine differs slightly from the algorithm above in that the BCD digits are read, and converted to binary, two digits at a time. Protection has also been added to detect BCD operands which, if converted, would yield binary values beyond the range of the result.

```plaintext
CONVERT_TO_BINARY
BIN:= 0
COUNT:= DIGNO
REPEAT
BCDACCUM:= BCDACCUM * 10
BIN:= 10 * BIN + CARRY DIGIT
COUNT:= COUNT - 1
UNTIL COUNT= 0
END CONVERT_TO_BINARY
```

This routine differs slightly from the algorithm above in that the BCD digits are read, and converted to binary, two digits at a time. Protection has also been added to detect BCD operands which, if converted, would yield binary values beyond the range of the result.

```plaintext
LOC OBJ SEQ SOURCE STATEMENT
1 $MACROFILE
2 INCLUDE( F1:CONBIN HED)
= 3 :*******************************************************************************
= 4 :*
= 5 :* CONBIN
= 6 :*
= 7 :*******************************************************************************
= 8 :*
= 9 :* THIS UTILITY CONVERTS A 6 DIGIT BCD VALUE TO BINARY
= 10 :* AT ENTRY
= 11 :* RD= POINTER TO A PACKED BCD STRING
= 12 :*
= 13 :* AT EXIT:
= 14 :* R = LOWER EIGHT BITS OF THE BINARY RESULT
= 15 :* XR= UPPER EIGHT BITS OF THE BINARY RESULT
= 16 :* C = SET IF OVERFLOW ELSE CLEARED
= 17 :*
= 18 :*******************************************************************************
19 :
20 :
21 $INCLUDE( F1:CONBIN PDL)
= 22 :
= 23 :
= 24 :1 CONVERT_TO_BINARY
= 25 :1 POINTER0:=POINTER0+DIGITPAIR-1
= 26 :1 COUNT:=DIGITPAIR
= 27 :1 BIN:=0
= 28 :1 REPEAT
= 29 :2 BIN:=BIN+10
= 30 :2 BIN:=BIN+MEM(RM7-4J
= 31 :2 BIN:=BIN+10
= 32 :2 BIN:=BIN+MEM(RM7-3-0]
```

All mnemonics copyrighted © Intel Corporation 1979. 1-69
LOC OBJ  SEQ  SOURCE STATEMENT

= 33 .2  POINTER0 :=POINTER0-1
= 34 .2  COUNT :=COUNT-1
= 35 .1  UNTIL COUNT=0
= 36 .1  END CONVERT_TO_BINARY
= 37
= 38  EODRATES
= 39. ======
= 40
0002  41  EA  EQU  R2
0003  42  COUNT  EQU  R3
0004  43  ICNT  EQU  R4
= 44
0005  45  Digpr  EQU  3
= 46
= 47  EJECT
= 48  $INCLUDE( F1.COMB1N)
= 49
= 50  TEMP1  SET  R5
= 51  TEMP2  SET  R6
= 52
= 53 .1  CONVERT_TO_BINARY
= 54  COMB1N:
= 55 .1  POINTER0 :=POINTER0+DIGITPAIR-1
= 56
0000  F8  = 56  MOV  A,R0
0001  0302  = 57  ADD  A,#DIGPR-1
0002  48  = 58  MOV  R0,A
= 59 .1  COUNT :=DIGITPAIR
= 60
0000  BB08  = 60  MOV  COUNT,#DIGPR
= 61 .1  BIN :=0
0006  27  = 62  CLR  A
0007  AA  = 63  MOV  #R,A
= 64 .1  REPEAT
= 65  CONBLP:
= 66 .2  BIN :=BIN+10
= 67
0000  F62A  = 68  JC  CONBER
= 69 .2  BIN :=BIN+MEM(R0)[7:4]
= 70
0000  AO  = 70  MOV  TEMP1,A
0000  F0  = 71  MOV  A,#R0
000E  47  = 72  SWAP  A
000F  530F  = 73  ANL  A,#0FH
0011  6D  = 74  ADD  A,TEMP1
0012  2A  = 75  XCH  A,AX
0013  1300  = 76  ADDC  A,#00
0015  2A  = 77  XCH  A,AX
0016  F62A  = 78  JC  CONBER
= 79 .2  BIN :=BIN+10
= 80
0018  F62A  = 80  CALL  CONB10
01A  F62A  = 81  JC  CONBER
= 82 .2  BIN :=BIN+MEM(R0)[3:0]
= 83
001C  AO  = 82  MOV  TEMP1,A
001D  F0  = 84  MOV  A,#R0
001E  530F  = 85  ANL  A,#0FH
0020  6D  = 86  ADD  A,TEMP1
0021  2A  = 87  XCH  A,AX
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0022</td>
<td>1300</td>
<td>88</td>
<td>ADDC A.000</td>
</tr>
<tr>
<td>0024</td>
<td>2A</td>
<td>89</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>0025</td>
<td>62A</td>
<td>90</td>
<td>JC CONSER</td>
</tr>
<tr>
<td>0027</td>
<td>C8</td>
<td>91</td>
<td>:2 POINDEX.0 =POINDEX.0-1</td>
</tr>
<tr>
<td>0028</td>
<td>E88</td>
<td>92</td>
<td>DEC R0</td>
</tr>
<tr>
<td>0029</td>
<td>83</td>
<td>93</td>
<td>COUNT = COUNT-1</td>
</tr>
<tr>
<td>002A</td>
<td>82</td>
<td>94</td>
<td>:1 UNTIL COUNT=0</td>
</tr>
<tr>
<td>002B</td>
<td>82</td>
<td>95</td>
<td>DNZ CONBLP</td>
</tr>
<tr>
<td>002C</td>
<td>82</td>
<td>96</td>
<td>:1 END CONVERT_TO_BINARY</td>
</tr>
<tr>
<td>002D</td>
<td>83</td>
<td>97</td>
<td>CONSER: RET</td>
</tr>
<tr>
<td>002E</td>
<td>82</td>
<td>98</td>
<td>= REJECT</td>
</tr>
<tr>
<td>002F</td>
<td>82</td>
<td>99</td>
<td>=:</td>
</tr>
<tr>
<td>0030</td>
<td>82</td>
<td>101</td>
<td>= UTILITY TO MULTIPLY BIN BY 10</td>
</tr>
<tr>
<td>0031</td>
<td>82</td>
<td>102</td>
<td>: CARRY WILL BE SET IF OVERFLOW OCCURS</td>
</tr>
<tr>
<td>0032</td>
<td>82</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>0033</td>
<td>82</td>
<td>104</td>
<td>CONB10: MOV TEMP1. A, SAVE A</td>
</tr>
<tr>
<td>0034</td>
<td>82</td>
<td>105</td>
<td>XCH A.XA ; SAVE XA</td>
</tr>
<tr>
<td>0035</td>
<td>82</td>
<td>106</td>
<td>MOV TEMP2. A</td>
</tr>
<tr>
<td>0036</td>
<td>82</td>
<td>107</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>0037</td>
<td>82</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>0038</td>
<td>82</td>
<td>109</td>
<td>CLR C</td>
</tr>
<tr>
<td>0039</td>
<td>82</td>
<td>110</td>
<td>PLC A ; BIN=BIN+2</td>
</tr>
<tr>
<td>003A</td>
<td>82</td>
<td>111</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>003B</td>
<td>82</td>
<td>112</td>
<td>RLC A</td>
</tr>
<tr>
<td>003C</td>
<td>82</td>
<td>113</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>003D</td>
<td>82</td>
<td>114</td>
<td>JC CONBIE ; ERROR ON OVERFLOW</td>
</tr>
<tr>
<td>003E</td>
<td>82</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>003F</td>
<td>82</td>
<td>116</td>
<td>RLC A ; BIN=BIN+4</td>
</tr>
<tr>
<td>0040</td>
<td>82</td>
<td>117</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>0041</td>
<td>82</td>
<td>118</td>
<td>RLC A</td>
</tr>
<tr>
<td>0042</td>
<td>82</td>
<td>119</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>0043</td>
<td>82</td>
<td>120</td>
<td>JC CONBIE ; ERROR ON OVERFLOW</td>
</tr>
<tr>
<td>0044</td>
<td>82</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>0045</td>
<td>82</td>
<td>122</td>
<td>ADD A. TEMP1 ; BIN=BIN+5</td>
</tr>
<tr>
<td>0046</td>
<td>82</td>
<td>123</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>0047</td>
<td>82</td>
<td>124</td>
<td>ADDC A. TEMP2</td>
</tr>
<tr>
<td>0048</td>
<td>82</td>
<td>125</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>0049</td>
<td>82</td>
<td>126</td>
<td>JC CONBIE ; ERROR ON OVERFLOW</td>
</tr>
<tr>
<td>004A</td>
<td>82</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>004B</td>
<td>82</td>
<td>128</td>
<td>RLC A ; BIN=BIN+10</td>
</tr>
<tr>
<td>004C</td>
<td>82</td>
<td>129</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>004D</td>
<td>82</td>
<td>130</td>
<td>RLC A</td>
</tr>
<tr>
<td>004E</td>
<td>82</td>
<td>131</td>
<td>XCH A.XA</td>
</tr>
<tr>
<td>004F</td>
<td>82</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>0050</td>
<td>83</td>
<td>133</td>
<td>CONBIE: RET</td>
</tr>
<tr>
<td>0051</td>
<td>83</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>0052</td>
<td>83</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>0053</td>
<td>83</td>
<td>136</td>
<td>END</td>
</tr>
</tbody>
</table>

USER SYMBOLS
CONB E0006 CONBIE E046 CONSER E02A CONBIN E0000 CONBLP E0006 COUNT E0003 DIGPR E0003 ICNT E0004 TEMP1 E0005 TEMP2 E0006 XA E0002

ASSEMBLY COMPLETE, NO ERRORS

All mnemonics copyrighted © Intel Corporation 1979.
CONCLUSION

The design goals of the full duplex serial communications software were realized; if transmission and reception are occurring concurrently, only 42 percent of the real time available to the 8049 will be consumed by the serial link. This implies that an 8049 running full duplex serial I/O will still outperform earlier members of the family running without the serial I/O requirement. It is also possible to run this program in an 8048 or 8748 at 1200 baud with the same 42 percent CPU utilization.

The execution times for the other routines that have been discussed have been summarized in Table 1. All of these routines were written to maintain maximum usability rather than minimum code size or execution time. The resulting execution times and code size are therefore what the user can expect to see in a real application. The results that were obtained clearly show the efficiency and speed of the 8049. The equivalent times for the 8048 are also shown. It is clear that the 8049 represents a substantial performance advantage over the 8048. Considering, in most applications, that the 8048 is the highest performance microcomputer available to date, the performance advantage of the 8049 should allow the cost benefits of a single chip microcomputer to be realized in many applications which up until now have required too much "computer power" for a single chip approach.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>8049</th>
<th>8048</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPY8</td>
<td>21</td>
<td>109</td>
</tr>
<tr>
<td>DIV 16</td>
<td>37</td>
<td>183 MIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>204 MAX</td>
</tr>
<tr>
<td>CONBCD</td>
<td>36</td>
<td>733</td>
</tr>
<tr>
<td>CONBIN</td>
<td>70</td>
<td>388</td>
</tr>
</tbody>
</table>

Table 1. Program Performance
A High-Speed Emulator
for Intel MCS-48
Microcomputers

August 1979
I. PURPOSE AND SCOPE

This Application Note presents a description of the design and operation of a high-speed emulator for the Intel® MCS-48™ family of single chip microcomputers. The HSE-49™ emulator provides a simple and inexpensive means for executing and debugging 8049 programs which require the full 11-MHz operating speed of the part.

Section II of this Application Note describes some of the features of this development tool and how it may be used. Section III briefly discusses the hardware used to implement these features, while Section IV describes the manner in which program execution status is made available to the operator.

A detailed description of all of the operator commands, is presented in Section V of this note, along with the modifiers and options which may be specified for each command. Known restrictions and limitations of the HSE-49 system are listed and explained in Section VI. Section VII shows how the basic circuit may be modified to provide options on memory organization, I/O configurations, etc.

Full schematics of the system hardware, as well as monitor software listings, are presented in Appendices A and B, respectively. A short summary of the command syntax is presented in Appendix C. Appendix D explains the error message codes which may appear during use.

It is assumed that the reader is already familiar with the operation of the 8048 or 8049 microcomputers. Some knowledge of the 8048 architecture is needed to understand sections of the command and modifier descriptions. Most users will already have this background. Other readers are referred to the MCS-48 Microcomputer User's Manual, Intel publication number 9800270.

II. THE HSE-49 DEVELOPMENT TOOL

In essence, the HSE-49 emulator provides the user a means for executing an MCS-48 program located in external RAM rather than internal ROM or EPROM. This allows programs being debugged to be modified easily and quickly during the debug cycle. A user's program may be entered into system RAM either manually or via a serial link from a host computer such as an Intel® Microcomputer Development System. Once loaded, the program can be modified using an on-board keyboard and display, and executed in real-time in a number of breakpoint modes. The internal state of the processor, including RAM, accumulator, timer/counter, and status register contents, can also be read and modified through the keyboard.

Breakpoint and debug facilities are extremely flexible. The following execution modes are provided.

- Programs may be run in full (11 MHz) real time;
- Programs may be single-stepped;
- In break mode, programs run in full real time until break occurs;
- Breaks may be triggered by either program or external data RAM accesses;
- Any number of breakpoints may be used in any combination;
- "Auto-Step" operation causes the current program counter and Accumulator contents to be printed on the display for a short time on every instruction cycle;
- "Auto-Break" provides the above display only when a break flag is encountered, with real time operation otherwise;
- While running in non-break mode, a TTL-level pulse is generated whenever a break flag is encountered. This signal may be used to trigger an oscilloscope or Logic Analyzer to assist in hardware and software debug.
- While running in any mode, the keyboard and display are “alive”. Execution may be suspended or terminated by commands from the keyboard.

Intent of this Note

While the HSE-49 emulator can assist a new microcomputer user in becoming familiar with the 8048 and 8049 microcomputers, its inherent debug capabilities will also prove helpful to design engineers. The design could be used for new system development and verification or adapted for prototype production.

The main concern in designing the HSE-49 emulator was to keep the basic design simple, while maximizing the system's flexibility. The design allows the use of jumpers, hardware and software switches, etc. to allow the user to reconfigure the system according to the way he dedicates chip-select pins, I/O, etc. The emulator can be changed to fit each user's unique needs, rather than forcing the user to alter his needs to what is provided.

The primary intent of note is to provide the reader with the information needed to reconstruct and make full use of the HSE-49 emulator. Less emphasis is placed on describing how the hardware operates or how the commands are implemented. This information may be found in the schematic diagrams and software listings included in the Appendices.

III. GENERAL HARDWARE OVERVIEW

User Program Emulation

The actual emulation of the user's program is done using an 8039 microcomputer (IC29 on the schematics in Appendix A) executing a program stored in external RAM. The basic minimum configuration includes the 8039 microcomputer, an 8282 address latch (IC19), and 2K bytes of 2114 RAM to use for program development and real-time execution (ICs B1, C1, B2, and C2). Additional RAM may be added to allow the user to expand his program and data memory to 4K each. (If an 11-MHz crystal is used with the microcomputer, type 2114-3 RAMs must be used.)
System Supervision

A second microcomputer — another 8039 (IC25) with an 8282 address latch (IC16) and off-chip program memory in a 2716 EPROM (IC15) — is used to scan the on-board keyboard and display, interpret and implement commands, drive serial interfaces, etc. In general, the master processor is used to interface the execution processor’s memory spaces with the outside world and control the operation of the execution processor. In this note the two processors will be abbreviated “MP” and “EP”, respectively. Figure 1 shows how the two processors interrelate with the rest of the system.

Keyboard/Display

The 33-key keyboard shown in Figure 2 includes a 16-key hexadecimal keypad and 17 special function keys for specifying commands and modifiers. Readers already familiar with the PROMPT-48™ debug tool for the 8048 will find that 25 of the HSE-49 emulator keys are identical in function and layout to the PROMPT-48 keyboard, and use the PROMPT-48 command syntax. The eight additional keys are used to generalize and augment the PROMPT-48 capabilities, as described in Section V.

The eight-character seven-segment display (DS1-DS8) is used for displaying addresses, data, and pseudo-alphanumeric messages. The display responses printed in Section V and throughout this note use a mix of upper and lower case letters to indicate what seven-segment patterns appear. An 8243 (IC9) and eight DIP packages (resistor packs, current buffers, etc.) are used for multiplexing the display and scanning the keyboard.

Breakpoint Detection

Breakpoints are specified and detected using a 2102A 1K x 8 RAM corresponding to each pair of 21145 (ICs A1

---

**Figure 1.** HSE-49™ Emulator Signal Flow Diagram

**Figure 2.** HSE-49™ Emulator Command Keyboard Organization
and A2). In effect, each program or data address accesses a 9-bit word. Eight bits are used normally for code or data storage. The ninth bit, accessed in parallel with the other eight, is used to indicate if a breakpoint has been set for that address. This output, when asserted, is latched (IC27 and IC36) and used to halt the execution processor via the single-step input. (In other modes, the break logic can be reconfigured to set the break requested flip-flop on any EP machine cycle or any EP “MOVX” instruction.)

**Link Register**

An 8212 8-bit latch (IC18) is used to communicate data and commands between the master and control processors. Under control of the MP, this register, called the “Link” register, may be logically mapped into either the program or data RAM address spaces. When this is done, the 2114s in the respective memory space are disabled and the link responds to all accesses, regardless of address. The link will be discussed in greater detail in Section IV.

**Control Logic**

In addition to the devices mentioned above, the minimum configuration requires about 10 additional ICs for bus arbitration, system control, and breakpoint and single-step logic. Additional parts may be optionally added for serial port interfacing, I/O reconstruction, etc.

**MP Monitor**

The monitor program executed by the MP includes commands for filling, reading, or writing the various memory spaces, including the execution processor’s program RAM, external (“MOVX”) data RAM, accumulator, PSW, PC, timer/counter, working registers, and internal RAM; to execute the user’s program from arbitrary addresses in various debugging modes; and to upload or download object or data files from diskettes using an Intel® development system. No special software is needed for the Intellec® other than ISIS Version 3.4 or later. The data format is compatible with the standard Intel hex file format produced by ASM-4: the baud rate may be altered from 110 baud (default state) up to 2400 baud from the on-board keybad. Blocks of data may be transmitted to a CRT or printer and displayed in a tabular format.

**IV. INTERPROCESSOR COMMUNICATION**

**Program Break Sequence**

When the MP detects that the EP has been halted by the breakpoint hardware, or when the operator presses a key while the program is executing, the program break sequence is initiated. The low-order 23 bytes of user program memory is read into a buffer within the internal RAM of the MP. A short program for reading and transmitting internal EP status is written over the low-order program memory. (This is one of several “mini-monitors” overlayed over the user program area.) The link register is mapped logically over the user program memory, and loaded with the 8049 machine code for a “CALL” instruction to the mini-monitor program area. The EP is then allowed to fetch a single instruction from the link, i.e., the “CALL” to the mini-monitor is forced onto the EP data bus.

From this point on, the EP executes code contained in the mini-monitor. The link is logically mapped over the data RAM address space (whether or not any 2114 data RAMs are present). A block diagram of the system at this point is shown in Figure 3. The break logic is reconfigured so that any “MOVX” (RD or WR) operation executed by the EP will cause it to halt.

For example, after entering the first mini-monitor, the EP executes a “MOVX @R0,A” instruction. This writes the contents of the accumulator prior to the execution termination into the link, and causes the EP to halt. The MP may then read and retain the link contents to determine the EP accumulator value. The EP timer/counter and PSW are preserved in the same manner.

**Accessing EP Internal RAM**

After reading and saving EP internal status, the MP loads a different mini-monitor into the same RAM area. This monitor allows the internal RAM of the EP to be read and written by the MP by passing address and data.
values between the two processors using the link register.

This is needed for two reasons. First, the EP program counter prior to the forced "CALL" instruction may be derived from the EP stack contents, and may be modified to cause the EP to resume execution at any desired address. Secondly, the internal RAM of the EP may then be accessed and modified in the process of executing a number of the monitor commands.

**Resuming User Program Execution**

In order to resume user program execution, a status-restoration mini-monitor is overlayed. This restores the EP internal status using a scheme analogous to the one in which the status was originally saved. The final step of the last mini-monitor is an "RETR" instruction, after which the EP is again halted. The low-order program memory saved earlier is rewritten into the appropriate area, the break logic is reconfigured for the desired execution mode, and the EP is released to run at full speed until the next break situation is encountered.

Note that all commands are implemented using "logical" rather than "physical" addressing. Thus the operator need not be concerned with the intricacies of the system design. For example, when any monitor command refers to low-order user program memory, the appropriate byte of storage within the MP internal RAM is accessed instead. If the location is altered, the internal RAM is modified appropriately. When program memory is reloaded prior to resuming user program execution, the modified version of the user program will be the one loaded.

**Data Manipulation/Control Command Group**

Commands:

* [EXAM/CHA]*

Display Response — "ECh."

Function — Examine/change memory location.

Causes the memory address specified to be read and presented on the display. New data may be entered (if desired) from the hexadecimal keypad. New data is verified before appearing on the display. Subsequent or previous locations may be read by pressing the [NEXT/] or [PREV] keys, respectively. Command terminated with [END/] key.

* [FILL]*

Display Response — "FIL."

Function — Fill range of memory addresses with a single data value.

Fill the appropriate memory space between the addresses specified by the first two parameters with the low-order byte of the third parameter. If second parameter less than first, only the location specified by the first is affected. If third parameter omitted, zero is assumed. If second and third parameters omitted, individual address specified is cleared. Command is useful for setting a large range of breakpoints; e.g., all of page 3 may be enabled for break with the command:

\[
[FILL][PROG BRK]<300>[,]<3FF>[,]<1>[,]
\]
[LIST]
Display Response — “LSt.”
Function — List memory to output device through HSE-49 serial port.
Display the contents of a range of addresses given by two parameters to a teletype or CRT screen. Data is formatted, 16 separated bytes per line, with the starting address of each line printed. If used with an Intellec® system, the operator first uses ISIS-II to transfer the TTY input to the CRT output (“COPY :TI: TO :CO:”) then invokes this command from the keypad. Alternatively, any ISIS device or disk file name (TO:, LP:, F:HRDREG.SAV, etc.) may be used as the destination.

[DNLOAD]
Display Response — “dnL.”
Function — Download memory through HSE-49 serial port.
Load data in hex file format through the serial input port. If used with Intellec® system, the operator first invokes this command from the keypad, then uses ISIS-II to transfer a disk file to the teletype port (“COPY :Fn:file.HEX TO :TO:”).
The use of the checksum field for the download command is expanded slightly. The Intel hex file format standard. If the first character of the checksum field is a question mark (“?”) the checksum for that record will not be verified. This allows large object files produced by the assembler to be patched using the ISIS text editor without the necessity of manually recomputing the checksum value.

[UPLOAD]
Display Response — “UPL.”
Function — Upload memory through HSE-49 serial port.
Output the contents of a range of addresses specified by two parameters through the HSE-49 serial port in standard Intel hex file format. If used with Intellec® system, the operator first uses ISIS-II to transfer the TTY input to a disk file (“COPY :TI: TO :Fn:file.HEX”), then invokes this command from the keypad.

Data types allowed:

[PROG MEM]
Display Response — “Pr.”
Function — User program memory.
Memory used to develop and execute user program.
Addresses 000 through 7FF are the execution processor’s memory bank 0; 800 through FFF are memory bank 1.

[REGISTER]
Display Response — “rG.”
Function — Register memory and RAM.
Internal RAM of execution processor. Locations 0-7 are working register bank 0; 18-1F are working register bank 1. Only the low-order 7 bits of an address are considered.

[DATA MEM]
Display Response — “dA.”
Function — External data memory (if installed).
Memory accessed by execution processor “MOVX A,@Rr” or “MOVX @Rr,A” instructions. High-order 4 bits may or may not be relevant, depending on jumpering option selected (explained in Section VII of this note).

[HARD REG]
Display Response — “Hr.”
Function — Hardware registers.
The execution processor (EP) hardware registers (accumulator, timer/counter, etc.), as well as several parameters for controlling HSE-49 status, are accessible through this catch-all memory space. Addresses are as follows:
00 — EP accumulator.
01 — EP PSW.
Bits correspond to 8049 PSW except that bit 3 (unused in the 8049) is used to monitor and alter the state of F1. Bits 2-0 correspond to the stack pointer value after the EP executes a CALL to the mini-monitor; i.e., one greater than when EP was running the user’s program.
02 — EP timer/counter.
03 — EP internal RAM location 00. (This value is also accessible through [REGISTER] space.)
04 — EP program counter (low byte).
05 — EP program counter (high nibble).
06-07 — HSE-49 serial interface baud rate parameters. Defaults to 110 baud; other rates may be selected by loading the values listed in Table 1.
08 — HSE-49 automatic sequencing rate parameter. Used in [GO][AUTO STP] and [GO][AUTO BRK] execution commands. 00 → fastest; FF → slowest. Defaults to 20H; approximately two steps per second.
09 — Monitor version/release number (packed BCD).
0A-0F — Currently unused by the monitor program.
10-7F — Variables used by master processor (MP) monitor. Should not be altered by operator.

[PROG BRK]
Display Response — “Pb.”
Function — User program breakpoint memory.

Memory space used to indicate points where program execution should halt when running in a mode with breakpoints enabled ([GO][W/ BRK] and [GO][AUTO BRK]). Break will occur if enabled byte is read as the first or last byte of a 2-byte instruction, or read in executing a MOV, MOV, or JMPP instruction. Memory is only 1 bit per location; 00 indicates continue, 01 causes a halt. Addresses 000 through 7FF are the execution processor’s memory bank 0; 800 through FFF are memory bank 1.

[DATA BRK]
Display Response — “db.”
Function — External data RAM breakpoint memory.

Memory space used to indicate points where data accesses should halt when running in a mode with breakpoints enabled ([GO][W/ BRK] and [GO][AUTO BRK]). Memory is only 1 bit per location; 00 indicates continue, 01 causes a halt. High-order 4 bits of breakpoint address may or may not be relevant, dependent on jumpering option selected for the corresponding data RAM (explained in Section VII of this note).

User Program Execution Control Group

Commands:

[GO]
Display Response — “Go.”
Function — Begin execution.

If a parameter is given as part of the command string, execution will begin at that address. Otherwise, the EP program counter (hardware registers 04 and 05) will be used. These will contain the program counter from an earlier program execution break unless they have since been explicitly modified by the operator.

If command is terminated by [END/], the EP’s F1, PSW and stack pointer will be cleared. If command string is terminated by [NEXT/], PSW will be taken from the EP PSW contents (hardware register 01).

While running the user’s program, the characters “—run—” are written on the display. Execution may be halted and another command initiated by pressing the appropriate command key. Execution may be suspended at any time in any mode by pressing the [END/] key. This will cause the current value of the execution processor program counter and accumulator appearing on the display in the form “PC.234-56”. System status is saved in the appropriate Hardware Registers. At this point, [NEXT/] will cause the program to execute one more instruction, or any other command may be invoked by pressing the appropriate command string. Does not affect the state of the Breakpoint Memories.

[DATA BRK]
Display Response — “db.”
Function — Go from reset state.

EP is hardware-reset and released to execute the user’s program from location 000H. No parameters are allowed. F0, F1, PSW, stack printer, memory bank flip-flop, etc., are cleared.

Note that this command does not require the use of mini-monitors to initiate program execution. As the last phase of the program development cycle, the 2114 program RAMs and address decoder may be removed and replaced by a ROM or EPROM part (not shown in schematics). This command may be used to start execution when the program RAM has been removed. No interrogation of EP status or internal RAM may be done, nor are break or single-step modes allowed in this case, though the 2102A breakpoint RAM outputs may still be used to trigger a logic analyzer.

Execution modes allowed:

[NO BRK]
Display Response — “nb.”
Function — Without breakpoints.

Full-speed execution without breakpoints enabled. Does not affect the state of the breakpoint memories.

[SING STP]
Display Response — “SSt.”
Function — Single Step.

Step through program one instruction at a time. After each instruction is executed, execution halts with the current value of the Execution Processor Program Counter and Accumulator appearing on the display in the form “PC.234-56”. System status is saved in the appropriate Hardware Registers. At the point, [NEXT/] will cause the program to execute one more instruction, or any other command may be invoked by pressing the appropriate command string. Does not affect the state of the Breakpoint Memories.

[W/ BRK]
Display Response — “br.”
Function — With breakpoints.

Full-speed execution with breakpoints enabled. When a breakpoint is encountered, execution halts with the current value of the execution processor program counter and accumulator appearing on the display in the form “PC.234-56”. System status is saved in the appropriate hardware registers. At this point, [NEXT/] will cause the program to continue until the next breakpoint is reached, or any other command may be invoked by pressing the appropriate command string.

[DATA BRK]
Display Response — “db.”
Function — Automatically sequence through a series of instructions.
Step through program one instruction at a time. After each instruction is executed, execution halts with the current value of the execution processor program counter and accumulator appearing on the display in the form "PC:234-56." System status is saved in the appropriate hardware registers. Execution resumes after a time determined by contents of hardware register 08. Does not affect the state of the breakpoint memories.

[AUTO BRK]
Display Response — "Abr."
Function — Automatically sequence between breakpoints.

Execute a series of instructions in real time between breakpoints. When breakpoint is encountered, halt EP temporarily while program counter and accumulator contents are displayed, then continue. Display is sustained after execution resumes. Does not affect the state of the breakpoint memories.

Breakpoint Control Command Group

Commands:
[AUTO BRK]
Display Response — "Abr."
Function — Automatically sequence between breakpoints.

Execute a series of instructions in real time between breakpoints. When breakpoint is encountered, halt EP temporarily while program counter and accumulator contents are displayed, then continue. Display is sustained after execution resumes. Does not affect the state of the breakpoint memories.

Breakpoint Control Command Group

Commands:

[B]
Display Response — "Stb."
Function — Breakpoint set.

Set breakpoint for the address given. Multiple breakpoints may be set by entering additional addresses, separated by the [NEXT] key. Command terminated by pressing [END/]. Action taken is to fill the appropriate breakpoint memory locations with logical ones.

[C]
Display Response — "CLb."
Function — Clear breakpoint.

Clear breakpoint for the address given. Multiple breakpoints may be cleared by entering additional addresses, separated by the [NEXT] key. Command terminated by pressing [END/]. Action taken is to fill the appropriate breakpoint memory locations with logical zeroes.

Data types allowed:

[PROG MEM]
Display Response — "Pr."
Function — Break on program memory fetch.

Applies command to the program breakpoint memory space.

[data MEM]
Display Response — "dA."
Function — Break on data memory access.

Applies command to the external data breakpoint memory space.

System Control Command Group

Command:

[SYS RST]
Display Response — "HSE-49."
Function — System reset.

Reset both the MP and EP and clear all breakpoints (requires approximately one second). CAUTION — If reset while EP is executing the user's program, the low order section of program memory (about 23 bytes) will be altered.

VI. SYSTEM LIMITATIONS

In designing the HSE-49 emulator, certain compromises were made in an attempt to maximize the usefulness of the emulator while keeping the circuitry simple and inexpensive. As a result, the following limitations exist and must be taken into account when using the system.

1. As explained in Section IV, user program execution is terminated (by single-stepping, breakpoints, pressing the [END/] key, etc.) by forcing the execution processor to execute a "CALL" instruction to the mini-monitor. This uses one level of the EP subroutine stack. The EP PSW reflects the value of the stack pointer after processing this CALL. As a result, the value indicated for stack depth by examining the EP PSW (hardware register 01) is one greater than the depth when the break was initiated. The user program must not be using all eight levels of stack when a break is initiated or the bottom level will be destroyed.

2. User program is initiated (by the [GO] command or when resuming execution after a breakpoint, single-stepping, etc.) by forcing the EP to execute an "RETR" instruction. This will clear the EP interrupt-in-progress flip-flop. If the user program allows both external and timer interrupts to be enabled at the same time, care must be taken to avoid causing a break while the EP is within an interrupt servicing routine. No limitation is placed on breakpoints or single-stepping in the background program because of this.

3. When the user program execution is terminated (by a break, single-stepping, etc.) and later resumed, the EP timer/counter is restored to its value when the break occurred (unless modified by the user). The prescaler, however, will have changed. Thus, up to 31 machine cycles may be "lost" or "gained" if a break occurs while the timer is running.

4. Timer interrupts occurring at the same time as an EP break may be ignored if the timer overflow occurs after breaking user program execution before the timer value is saved.

5. The 8049 "RET" and "RETR" instructions are each 1-byte, 2-cycle instructions. During the second cycle the byte following the return instruction is fetched and ignored. If a program breakpoint is set for a location following a "RET" or "RETR" instruction, a break will be initiated when the return is executed.
6. Breakpoints should not be placed in the last 3 bytes of an EP memory bank (locations 7FDH-7FFH and 0FFDH-0FFFF). User program should not be single-stepped or auto-stepped through these locations.

7. Since I/O configuration is determined by external hardware rather than software, I/O modes may not be altered while a program is executing. (See Section VII for further details.)

8. The "ANL BUS,#nn" and "ORL BUS,#nn" instructions may not be used in the user program, as external hardware cannot properly restore these functions.

9. The memory bank select flag is not affected by the user program break sequence. Upon resuming execution with the [GO] command this flag will remain in the same state as before the preceding break. The flag may be cleared only by executing the [GO/RESET] or [SYS RST] commands.

VII. HARDWARE CONFIGURATIONS

A number of control and status lines are available to the user. All are low-power Schottky TTL-compatible signals.

TP1 — Unused MP input.
TP2 — Unused MP output.
TP3 — User program suspended. Low when EP running user code. High when halted or running mini-monitors.
TP4 — Breakpoint encountered. Normally low. High-level pulse generated when breakpoint passed. Useful for triggering logic analyzers, oscilloscopes, etc.
TP5 & TP6 — Memory matrix mode control. Select program vs. data RAM, link mapping configuration, etc. (See Appendix B for details.)
TP7 — Bus control. Low when MP controls common memory buses. High when EP controls memory buses.

The HSE-49 emulator hardware is designed to allow the user to reconfigure the system for a wide variety of different applications by installing or removing jumper wires or additional components. The schematics in Appendix A show the components needed for a variety of different configurations. In general, not all of the devices are required (or allowed) for any one configuration. The devices which are required are included in the following description.

The types of options allowed are divided below into several general classes and subdivided into mutually-independent features. Within some of these features there are numbered, mutually exclusive configurations; i.e., the serial interface (if desired) may use either current-loop or RS-232C current buffers, but not both at one time.

Standard Operating Configuration

(Minimum system configurations — up to 4K program RAM; no data RAM; no serial interfaces; no execution processor I/O reconstruction.)

A. Basic 2K monitor from Appendix B:
   - Install resistors R4-R6
   - Install transistor Q1
   - Install crystals Y1-Y2
   - Install capacitors C5-C38
   - Install switches S1-S33
   - Install displays DS1-DS8
   - Install IC1-IC2
   - Install RP3-RP5
   - Install IC6-IC7
   - Install RP8
   - Install IC9
   - Install IC15-IC20
   - Install IC25-IC30
   - Install IC34
   - Install IC36-IC38
   - Install A1-A2
   - Install B1-B2
   - Install C1-C3
   - Install jumpers 13-15
   - Install jumpers 17-18
   - Install jumper 20

B. Expansion 2K monitor:
   - Install IC14
   - Remove jumper 17

Serial Interface Buffer Selection

A. Current loop serial interfaces (4N46s) installed for use with full Intellec® Model 800 development system TTY port.
   - Install IC21-IC22
   - Install resistor R1-R3
   - Install jumpers 4-9
   (Remove current-loop jumpers)

B. RS-232C serial interfaces (MC1488 and MC1489) installed for use with CRT as output device for data dumps:
   - Install IC23-IC24
   - Install jumpers 1-3
   - Install jumpers 10-11
   (Remove current-loop jumpers)

External Data RAM Address Decoding Scheme for Execution Processor

A. Up to 16 pages of on-board external data RAM installed for execution processor (addresses 0 through
0FFFH = 4K bytes); port 2 used for addressing pages 0 through 15:
Install jumpers 21–25
Install jumper 27
Install A5–A8
Install B5–B8
Install C5–C8

B. One page of on-board external data RAM installed for execution processor (addresses 0 through 0FFH); port 2 not used for data addressing:
Install jumper 26
Install jumper 28
Install A5
Install B5
Install C5
Connect the outputs of IC20, pins 7, 9, 10, & 11 to the inputs of a 74LS21 AND gate (not shown). Connect the output to CE and CS inputs of A5–C5. (Note: these signals are all present at jumpers 21–24 on the schematics.)

Reconstructing I/O for Execution Processor

A. Application of port 2, pins P23–P20:
(1) Using P23–P20 for latched output data (used with “OUTL P2,A”, “ANL P2,#data”, and “ORL P2,#data” instructions):
Install IC31
(2) Using P23–P20 for interfacing to an 8243 in user's prototype:
Connect D3–D0 pins on IC31 socket to corresponding Q3–Q0 pins.

B. Application of execution processor BUS:
(1) Use of BUS as latched output port (“OUTL BUS,A”):
Install IC32
1 $NUMER IMAGE XREF
2 TITLE: HSE-49(TM) EMULATOR MONITOR VERSION 2.5
3
4:=========================================================================
5
6 : PROGRAM: HSE-49(TM) EMULATOR MONITOR
7 : VERS 2.5/76
8 :
9 : COPYRIGHT (C) 1976
10 : INTEL CORPORATION
11 : 3885 DOORS AVENUE
12 : SANTA CLARA, CALIFORNIA 95051
13 :
14 :=========================================================================
15 :
16 : ABSTRACT
17 :
18 : HIS PROGRAM CONTAINS THE SOFTWARE NECESSARY TO RUN THE HSE-49(TM)
19 : HIGH-SPEED EMULATOR FOR INTEL'S MCS-48(TM) FAMILY FAMILY OF MICROCOMPUTERS.
20 : THE EMULATOR PROVIDES AN ASSORTMENT OF UTILITY FUNCTIONS FOR
21 : DEVELOPING AND DEBUGGING 8049-BASED APPLICATIONS, INCLUDING THE
22 : ABILITY TO ENTER AND MODIFY PROGRAMS IN PROGRAM RAM.
23 : OVER DATA SINGLE-STEP EXECUTION OF A PROGRAM AND EXECUTE PROGRAMS
24 : AT SPEEDS OF UP TO 11 MIPS WITH OR WITHOUT BREAKPOINTS ENABLED.
25 : THE EMULATOR IS DESCRIBED IN DETAIL IN INTEL'S APPLICATION NOTE
26 : AP-55: "A HIGH-SPEED EMULATOR FOR INTEL MCS-48(TM) MICROCOMPUTERS."
27 :
28 : PROGRAM ORGANIZATION
29 : ==
30 ::
31 : THIS LISTING IS ORGANIZED AS FOLLOWS:
32 :
33 : : INTRODUCTION AND HARDWARE OVERVIEW;
34 : : VARIABLE DECLARATION AND DEFINITION;
35 : : POWER-ON SYSTEM INITIALIZATION;
36 : : KEYBOARD COMMAND PARSER AND ASSOCIATED TABLES;
37 : : IMPLEMENTATIONS OF THE PRIMARY COMMANDS;
38 : : DATA ACCESSING UTILITY SUBROUTINES USED THROUGHOUT;
39 : : KEYBOARD SCANNING AND DISPLAY DRIVING SUBROUTINE;
40 : : KEYBOARD AND DISPLAY INTERFACING UTILITIES;
41 : : ROUTINES AND UTILITY SUBROUTINES WHICH INTERACT BETWEEN NP AND LP.
42 :
43 :
44 :
45 : REJECT

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td></td>
<td></td>
<td><strong>INTRODUCTION AND HARDWARE OVERVIEW</strong></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td><strong>----------</strong></td>
</tr>
<tr>
<td>49</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td><strong>THE EMULATOR DESIGN USES TWO MICROPRIORCESSORS. ONE PROCESSOR CONTROLS</strong></td>
</tr>
<tr>
<td>51</td>
<td></td>
<td></td>
<td><strong>SYSTEM STATUS, INTERPRETS MONITOR COMMANDS, AND COMMUNICATES</strong></td>
</tr>
<tr>
<td>52</td>
<td></td>
<td></td>
<td><strong>WITH THE OUTSIDE WORLD THROUGH THE ON-BOARD KEYBOARD, DISPLAY, SERIAL</strong></td>
</tr>
<tr>
<td>53</td>
<td></td>
<td></td>
<td><strong>INTERFACES, CONTROL SIGNALS, ETC.</strong></td>
</tr>
<tr>
<td>54</td>
<td></td>
<td></td>
<td><strong>A SECOND PROCESSOR IS USED TO ACTUALLY</strong></td>
</tr>
<tr>
<td>55</td>
<td></td>
<td></td>
<td><strong>EXECUTE THE USER'S PROGRAM UNDER THE CONTROL OF THE FIRST.</strong></td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td><strong>THESE PROCESSORS ARE REFERRED TO</strong></td>
</tr>
<tr>
<td>57</td>
<td></td>
<td></td>
<td><strong>THROUGHOUT THIS PROGRAM AS THE MASTER PROCESSOR (MP) AND EXECUTION</strong></td>
</tr>
<tr>
<td>58</td>
<td></td>
<td></td>
<td><strong>PROCESSOR (CP) RESPECTIVELY.</strong></td>
</tr>
<tr>
<td>59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td><strong>THE PROGRAM IN THIS LISTING IS EXECUTED BY THE MASTER PROCESSOR.</strong></td>
</tr>
<tr>
<td>61</td>
<td></td>
<td></td>
<td><strong>AT THE END OF THIS LISTING ARE SEVERAL SHORT &quot;MINI-MONITOR OVERLAYS&quot;</strong></td>
</tr>
<tr>
<td>62</td>
<td></td>
<td></td>
<td><strong>WHICH THE EXECUTION PROCESSOR EXECUTES WHEN INTERACTION BETWEEN THE</strong></td>
</tr>
<tr>
<td>63</td>
<td></td>
<td></td>
<td><strong>TWO PROCESSORS IS NECESSARY</strong></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td></td>
<td></td>
<td><strong>THIS PROGRAM WAS WRITTEN USING A NUMBER OF MACROS TO HANDLE THE ALLOCATION</strong></td>
</tr>
<tr>
<td>66</td>
<td></td>
<td></td>
<td><strong>OF MP RESOURCES (WORKING REGISTERS, INTERNAL RAM, AND MP MONITOR ROM</strong></td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
<td><strong>FOR CODE AND DATA STORAGE). THESE MACRO DEFINITIONS ARE INCLUDED IN A FILE</strong></td>
</tr>
<tr>
<td>68</td>
<td></td>
<td></td>
<td><strong>NAMED &quot;ALLOC MAC,&quot; AND ARE PRINTED IN THIS LISTING FOR REFERENCE.</strong></td>
</tr>
<tr>
<td>69</td>
<td></td>
<td></td>
<td><strong>ANOTHER SET OF MACROS IS USED TO SIMPLIFY THE ACCESSING OF VARIABLES</strong></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td><strong>STORED IN INTERNAL RAM (AS OPPOSED TO WORKING REGISTERS) BY USING R1 TO</strong></td>
</tr>
<tr>
<td>71</td>
<td></td>
<td></td>
<td><strong>INDIRECTLY ADDRESS THE APPROPRIATE RAM LOCATION WHEN NECESSARY.</strong></td>
</tr>
<tr>
<td>72</td>
<td></td>
<td></td>
<td><strong>THESE MACROS ARE INCLUDED IN &quot;MOCODE MAC,&quot; AND ARE ALSO PRINTED HERE.</strong></td>
</tr>
<tr>
<td>73</td>
<td></td>
<td></td>
<td><strong>COMPLETE UNDERSTANDING OF THESE MACROS IS NOT REQUIRED TO UNDERSTAND THE</strong></td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
<td><strong>MONITOR PROPER. ALL LINES WHICH ACTUALLY PRODUCE OBJECT CODE APPEAR IN</strong></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
<td><strong>THE LISTING ITSELF, INDENTED TWO SPACES FROM THE NORMAL TABULATION COLUMNS.</strong></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
<td><strong>THE ACTUAL MONITOR PROGRAM FOR THE EMULATOR BEGINS AT APPROXIMATELY</strong></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
<td><strong>SOURCE LINE NUMBER 500.</strong></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
<td>**LINES GENERATED BY MACRO EXPANSION ARE FLAGGED BY A PLUS SIGN (+) **</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td><strong>IMMEDIATELY FOLLOWING THE SOURCE LINE NUMBER.</strong></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
<td><strong>A NUMBER OF LINES FROM THE VARIOUS MACRO DEFINITIONS WHICH DO NOT</strong></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
<td><strong>PRODUCE NON-OBJECT CODE ARE PROCESSED BY THE ASSEMBLER</strong></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
<td><strong>AS THESE MACROS ARE EXPANDED. WHEN THIS IS THE CASE, THESE LINES ARE</strong></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
<td><strong>SUPPRESSED FROM THE LIST FILE. AS A RESULT, THE LINE NUMBERS ARE</strong></td>
</tr>
<tr>
<td>85</td>
<td></td>
<td></td>
<td><strong>NOT ALWAYS CONSECUTIVE WHERE A MACRO IS BEING INVOKED.</strong></td>
</tr>
<tr>
<td>86</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td></td>
<td></td>
<td><strong>NOTE:</strong></td>
</tr>
<tr>
<td>88</td>
<td></td>
<td></td>
<td><strong>=====</strong></td>
</tr>
<tr>
<td>89</td>
<td></td>
<td></td>
<td>&quot;SOURCE-LINE&quot; REFERS TO THE DECEIMAL NUMBERS LEFT OF EACH INSTRUCTION**</td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td><strong>AT THE END OF THE LISTING IS AN ASSEMBLY CROSS-REFERENCE TABLE INDICATING</strong></td>
</tr>
<tr>
<td>91</td>
<td></td>
<td></td>
<td><strong>THE SEQUENTIAL SOURCE-LINE NUMBER OF ALL INSTANCES WHERE ANY VARIABLE</strong></td>
</tr>
<tr>
<td>92</td>
<td></td>
<td></td>
<td><strong>IS DEFINED OR REFERENCED. THIS WILL BE OF GREAT ASSISTANCE IN</strong></td>
</tr>
<tr>
<td>93</td>
<td></td>
<td></td>
<td><strong>LOCATING SPECIFIC SUBROUTINES, ETC. IN THE LISTING.</strong></td>
</tr>
<tr>
<td>94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td></td>
<td></td>
<td><strong>MNEMONICS COPYRIGHT © 1976 INTEL CORPORATION</strong></td>
</tr>
<tr>
<td>96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>97</td>
<td></td>
<td></td>
<td><strong>REJECT</strong></td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ | LINE | SOURCE STATEMENT
--- | --- | ---
0000 | 39 | INCLUDE ( PA.ALLOC. MAC )
0000 | = 39 | ?P1 EQU 0
0000 | = 100 :
0000 | = 101 ?R0 EQU 0
0001 | = 102 ?R1 EQU 1
0002 | = 102 ?R2 EQU 2
0003 | = 104 ?CONST EQU 3
0004 | = 105 ?R EQU 4 ; ACCUMULATOR VARIABLE TYPE
          = 106 ;
          = 107 ; THE FOLLOWING Initializes THE LINKED LIST POINTERS FOR
          = 108 ; THE REGISTER ALLOCATION AND DEALLOCATION ROUTINES.
          = 109 :
0003 | = 110 ?R0R2 SET 3
0004 | = 111 ?R0R3 SET 4
0005 | = 112 ?R0R4 SET 5
0006 | = 113 ?R0R5 SET 6
0007 | = 114 ?R0R6 SET 7
0008 | = 115 ?R0R7 SET 8
          = 116 ;
0002 | = 117 ?R0PNT SET 2
          = 118 ;
0003 | = 119 ?R1R2 SET 3
0004 | = 120 ?R1R3 SET 4
0005 | = 121 ?R1R4 SET 5
0006 | = 122 ?R1R5 SET 6
0007 | = 123 ?R1R6 SET 7
0008 | = 124 ?R1R7 SET 8
          = 125 ;
0002 | = 126 ?R1PNT SET 2
          = 127 ;
0000 | = 128 ORGP0 SET 000H
0100 | = 129 ORGP1 SET 100H
0200 | = 130 ORGP2 SET 200H
0300 | = 131 ORGP3 SET 300H
0400 | = 132 ORGP4 SET 400H
0500 | = 133 ORGP5 SET 500H
0600 | = 134 ORGP6 SET 600H
0700 | = 135 ORGP7 SET 700H
          = 136 ;
          = 137 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC 089  |  LINE  |  SOURCE STATEMENT

= 130                          ;**************************************************************************
= 139 :
= 140 ;                      START OF ALLOCATION MACROS
= 141 :
= 142 ;                          ;**************************************************************************
= 143 :
= 144 ?RSAVE MACRO SYMBOL BANK PNTVAL
= 145 IF PNTVAL EQ 0
= 146 ERROR 2
= 147 EXIT
= 148 ENDIF
= 149 $ ;  SAVE GEN
= 150 SYMBOL SET RAPNTVAL
= 151 $ ;  RESTORE
= 153 ENDM
= 154 ;
= 155 ;
= 156 ?MINDX SET 20H
= 157 ;
= 158 ?MCSAVE MACRO SYMBOL LENGTH ADDR
= 159 $ ;  SAVE GEN
= 160 SYMBOL EQU ADDR
= 161 $ ;  RESTORE
= 162 ?MINDX SET ?MINDX?LENGTH
= 163 ENDM
= 164 ;
= 165 MBLOCK MACRO SYMBOL LENGTH
= 166 ?$SYMBOLEQU 3
= 167 ?MCSAVE SYMBOL LENGTH ?MINDX
= 168 ENDM
= 169 ;
= 170 DECLARE MACRO SYMBOL TYPE
= 171 ?$SYMBOLEQU ?TYPE
= 172 IF ?TYPE EQ 2
= 173 ?MCSAVE SYMBOL 1 ?MINDX
= 1/4 EXIT
= 175 ENDIF
= 176 IF ?TYPE EQ 0
= 177 ?RSAVE SYMBOL 0 ?BOPNT
= 178 EXIT
= 179 ENDIF
= 180 IF ?TYPE EQ 1
= 181 ?RSAVE SYMBOL 1 ?DIPNT
= 182 EXIT
= 183 ENDIF
= 184 ENDM
= 185 ;
= 186 $ ;  EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ     LINE     SOURCE STATEMENT

= 107 ;
= 108 ;PEORG MACRO TO RESET THE INSTRUCTION LOCATION COUNTER
= 109 ; TO THE FIRST FREE LOCATION ON THE FIRST PAGE MODULE WILL
= 110 ; FIT WITHIN
= 111 ;PEORG MACRO LOCATION
= 112 ;SAVE GEN
= 113 ;ORG LOCATION
= 114 ;$RESTORE
= 115 ;ENDM
= 116 ;
= 117 ;CODEBLK MACRO TO FIND A PAGE OF ROM
= 118 ; WHICH THIS BLOCK OF CODE WILL FIT WITHIN
= 119 ;CODEBLK MACRO LENGTH
= 120 ;LENGTH SET LENGTH
= 121 ;IF HIGH(ORGPG0+LENGTH-1) EQ 0
= 122 ;REORG 20RGP0
= 123 ;START SET $
= 124 ;EXITM
= 125 ;ENDIF
= 126 ;IF HIGH(ORGPG1+LENGTH-1) EQ 1
= 127 ;REORG 20RGP1
= 128 ;START SET $
= 129 ;EXITM
= 130 ;ENDIF
= 131 ;IF HIGH(ORGPG2+LENGTH-1) EQ 2
= 132 ;REORG 20RGP2
= 133 ;START SET $
= 134 ;EXITM
= 135 ;ENDIF
= 136 ;IF HIGH(ORGPG4+LENGTH-1) EQ 4
= 137 ;REORG 20RGP4
= 138 ;START SET $
= 139 ;EXITM
= 140 ;ENDIF
= 141 ;IF HIGH(ORGPG5+LENGTH-1) EQ 5
= 142 ;REORG 20RGP5
= 143 ;START SET $
= 144 ;EXITM
= 145 ;ENDIF
= 146 ;IF HIGH(ORGPG6+LENGTH-1) EQ 6
= 147 ;REORG 20RGP6
= 148 ;START SET $
= 149 ;EXITM
= 150 ;ENDIF
= 151 ;IF HIGH(ORGPG7+LENGTH-1) EQ 7
= 152 ;REORG 20RGP7
= 153 ;START SET $
= 154 ;EXITM
= 155 ;ENDIF
= 156 ;IF HIGH(ORGPG3+LENGTH-1) EQ 3
= 157 ;REORG 20RGP3
= 158 ;START SET $
= 159 ;EXITM
= 160 ;ENDIF
= 161 ;ERROR 0 ;*** INSUFFICIENT SPACE FOR CODE ON ANY PAGE ***
LOC OBJ LINE SOURCE STATEMENT

= 242  ENDM
= 243  DB:TABLEK INSERTS INTO PAGE 3
= 244  DATABK MACRO LENGTH
= 245  ?LENGTH SET LENGTH
= 246  IF HIGH(ORGPG3+LENGTH-1) EQ 3
= 247  REORG XORGPG3
= 248  ?START SET $
= 249  EXIT
= 250  ENDF
= 251  ERROR 0 ;*** INSUFFICIENT SPACE FOR DATA BLOCK ON PAGE 3 ***
= 252  ENDM
= 253  SIZE PRINTS A LINE TO THE SOURCE FILE GIVING BLOCK SIZE.
= 254  AND UPDATES APPROPRIATE ORGP#.
= 255  SIZE MACRO BlK:PGE
= 256  SIZE G#E
= 257  SIZE SET BLK
= 258  
= 259  ;***********************************************************************
= 260  IF ?LENGTH LT SIZE
= 261  ERROR 0 ;*** SIZE EXCEEDS SPACE CHECKED FOR BY CODEBLK MACRO
= 262  ENDF
= 263  IF HIGH($-1) NE HIGH(?START)
= 264  ERROR 0 ;*** CODE OR DATA BLOCK ROLLED OVER PAGE BOUNDARY ***
= 265  ENDF
= 266  RESTORE
= 267  ORGP#GSE SET $
= 268  ENDM
= 269  SIZECK CHECKS SIZE OF PRECEDING BLOCK. PRINTS SIZE TO .LSI FILE
= 270  SIZECK MACRO
= 271  SIZE ($-$?START);HIGH(?START)
= 272  ENDM
= 273  ;
= 274  ;
= 275  RSOURCE CODE SPACE ALLOCATION SUMMARY STATEMENT
= 276  RSOURCE MACRO
= 277  $RAW LIST GEN
= 278  PGSIZE SET ORGPG8-000H ;BYTES USED ON PAGE 0
= 279  PGSIZE SET ORGPG1-000H ;BYTES USED ON PAGE 1
= 280  PGSIZE SET ORGPG2-000H ;BYTES USED ON PAGE 2
= 281  PGSIZE SET ORGPG3-000H ;BYTES USED ON PAGE 3
= 282  PGSIZE SET ORGPG4-000H ;BYTES USED ON PAGE 4
= 283  PGSIZE SET ORGPG5-000H ;BYTES USED ON PAGE 5
= 284  PGSIZE SET ORGPG6-000H ;BYTES USED ON PAGE 6
= 285  PGSIZE SET ORGPG7-000H ;BYTES USED ON PAGE 7
= 286  $EJECT
= 287  $RESTORE
= 288  ENDM
= 289  $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
290:
291 $ INCLUDE (:F0: MOPCOD.MAC)
292:
293 $FORM1 MACRO FOR GENERALIZING OPCODE INSTRUCTION
294:
295 $FORM1 MACRO OPCODE, SRC
296 IF $麝RC EQ 2
297 $ SAVE GEN
298 MOV R1,$麝RC
299 OPCODE R1, #R1
300 $ RESTORE
301 EXITM
302 ENDIF
303 IF $麝RC EQ 0 OR $麝RC EQ 1
304 $ SAVE GEN
305 OPCODE R1, #SRC
306 $ RESTORE
307 EXITM
308 ENDIF
309 IF $麝RC EQ 3
310 $ SAVE GEN
311 OPCODE R1, #SRC
312 $ RESTORE
313 EXITM
314 ENDIF
315 ERROR 1
316 ENDM
317
318 $FORM2 MACRO FOR GENERALIZING MOVES FROM THE ACC TO A VARIABLE
319 $FORM2 MACRO DEST
320 IF $麝EST EQ 2
321 $ SAVE GEN
322 MOV R1, #麝EST
323 MOV $R1, A
324 $ RESTORE
325 EXITM
326 ENDIF
327 IF $麝EST EQ 0 OR $麝EST EQ 1
328 $ SAVE GEN
329 MOV DEST, R
330 $ RESTORE
331 EXITM
332 ENDIF
333 ERROR 1
334 ENDM
335
336 $FORM3 MACRO FOR GENERALIZING MOVES FROM THE ACC TO A VARIABLE WHEN IT IS KNOWN THAT R1 (IF NEEDED FOR INDIRECT ADDRESSING) IS ALREADY PRESET
337 $FORM3 MACRO DEST
338 IF $麝EST EQ 2
339 $ SAVE GEN
340 MOV $R1, A
341 $ RESTORE
342 EXITM
LOC OBJ  |  LINE |  SOURCE STATEMENT
---|---|---
- | = 345 ENDIF | 
- | = 346 IF | ?DEST EQ 0 OR ?DEST EQ 1
- | = 347 $ | SAVE GEN
- | = 348 | MOV DEST A
- | = 349 $ | RESTORE
- | = 350 | EXITM
- | = 351 ENDIT | 
- | = 352 | ERROR 1
- | = 353 ENDM | 
- | = 354 | ;
- | = 355 | ;
- | = 356 | FORM4 MACRO FOR GENERALIZING MOV A, SRC INSTRUCTION
- | = 357 | IF | ?SRC EQ 2
- | = 358 $ | SAVE GEN
- | = 359 | MOV RL #SRC
- | = 360 | MOV A, #RL
- | = 361 $ | RESTORE
- | = 362 | EXITM
- | = 363 ENDIF | 
- | = 364 IF | ?SRC EQ 0 OR ?SRC EQ 1
- | = 365 $ | SAVE GEN
- | = 366 | MOV A, #SRC
- | = 367 $ | RESTORE
- | = 368 | EXITM
- | = 369 ENDIF | 
- | = 370 IF | ?SRC EQ 3
- | = 371 $ | SAVE GEN
- | = 372 | MOV A, #SRC
- | = 373 $ | RESTORE
- | = 374 | EXITM
- | = 375 ENDIF | 
- | = 376 | ERROR 1
- | = 377 ENDM | 
- | = 378 | ;
- | = 379 | ;
- | = 380 | FORM5 MACRO FOR GENERALIZING MOVING A CONSTANT INTO A VARIABLE
- | = 381 | IF | ?DEST EQ 0 OR ?DEST EQ 1 OR ?DEST EQ 4
- | = 382 $ | SAVE GEN
- | = 383 | MOV DEST, #CONST
- | = 384 $ | RESTORE
- | = 385 | EXITM
- | = 386 ENDIF | 
- | = 387 IF | ?DEST EQ 2
- | = 388 $ | SAVE GEN
- | = 389 | MOV RL, #DEST
- | = 390 | MOV @RL, #CONST
- | = 391 $ | RESTORE
- | = 392 | EXITM
- | = 393 ENDIF | 
- | = 394 | ERROR 1
- | = 395 ENDM | 
- | = 396 | ;
- | = 397 | ;
- | = 398 | MMOV MACRO GENERALIZED MOVE FROM SRC TO DEST
- | = 399 | MMOV MACRO DEST, SRC
- | = 399 IF | ?SRC EQ 3

All mnemonics copyrighted © Intel Corporation 1976.
LOC | DCJ | LINE | SOURCE STATEMENT
---|---|---|---
= | 400 | ?FORM5 DEST, SRC
= | 401 | EXITM
= | 402 | ENDM
= | 403 | IF ?DEST EQ 4
= | 404 | ?FORM1 MOV, SRC
= | 405 | EXITM
= | 406 | ENDM
= | 407 | IF ?SRC EQ 4
= | 408 | ?FORM2 DEST
= | 409 | EXITM
= | 410 | ENDM
= | 411 | ?FORM1 MOV, SRC
= | 412 | ?FORM2 DEST
= | 413 | ENDM
= | 414 | "BINOP MACRO GENERALIZES ARITHMETIC AND LOGICAL OPERATIONS"
= | 415 | "BINOP MACRO OF CODE, DEST, SRC"
= | 416 | IF ?DEST EQ 4
= | 417 | ?FORM1 OF CODE, SRC
= | 418 | EXITM
= | 419 | ENDM
= | 420 | IF ?SRC EQ 4
= | 421 | ?FORM1 OF CODE, DEST
= | 422 | ?FORM2 DEST
= | 423 | EXITM
= | 424 | ENDM
= | 425 | ?FORM1 MOV, SRC
= | 426 | ?FORM1 OF CODE, DEST
= | 427 | ?FORM2 DEST
= | 428 | ENDM
= | 429 | "MADD MACRO FOR GENERALIZING ADD INSTRUCTION"
= | 430 | "MADD MACRO DEST, SRC"
= | 431 | "BINOP ADD, DEST, SRC"
= | 432 | ENDM
= | 433 | ;
= | 434 | "MADD MACRO FOR GENERALIZING ADDC INSTRUCTION"
= | 435 | "MADD MACRO DEST, SRC"
= | 436 | "BINOP ADDC, DEST, SRC"
= | 437 | ENDM
= | 438 | ;
= | 439 | "MANL MACRO FOR GENERALIZING ANL INSTRUCTION"
= | 440 | "MANL MACRO DEST, SRC"
= | 441 | "BINOP ANL, DEST, SRC"
= | 442 | ENDM
= | 443 | ;
= | 444 | "MORL MACRO FOR GENERALIZING ORL INSTRUCTION"
= | 445 | "MORL MACRO DEST, SRC"
= | 446 | "BINOP ORL, DEST, SRC"
= | 447 | ENDM
= | 448 | ;
= | 449 | "MSPL MACRO FOR GENERALIZING XRL INSTRUCTION"
= | 450 | "MSPL MACRO DEST, SRC"
= | 451 | "BINOP XRL, DEST, SRC"
= | 452 | ENDM
= | 453 | ;
= | 454 | "MXCH MACRO FOR GENERALIZING XCH INSTRUCTION"

All mnemonics copyrighted © Intel Corporation 1976.

1-94
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>455</td>
<td>MACH</td>
<td>455</td>
<td>MACRO DEST, SRC</td>
</tr>
<tr>
<td>456</td>
<td>MACH</td>
<td>456</td>
<td>MACRO XCH, DEST, SRC</td>
</tr>
<tr>
<td>457</td>
<td>MACH</td>
<td>457</td>
<td>ENDM</td>
</tr>
<tr>
<td>458</td>
<td>MACH</td>
<td>458</td>
<td></td>
</tr>
<tr>
<td>459</td>
<td>MACH</td>
<td>459</td>
<td>MACRO OPCODE, DEST</td>
</tr>
<tr>
<td>460</td>
<td>MACH</td>
<td>460</td>
<td>FORM2 MOV, DEST</td>
</tr>
<tr>
<td>461</td>
<td>MACH</td>
<td>461</td>
<td>$SAVE GEN</td>
</tr>
<tr>
<td>462</td>
<td>MACH</td>
<td>462</td>
<td>OPCODE A</td>
</tr>
<tr>
<td>463</td>
<td>MACH</td>
<td>463</td>
<td>$RESTORE</td>
</tr>
<tr>
<td>464</td>
<td>MACH</td>
<td>464</td>
<td>FORM3 DEST</td>
</tr>
<tr>
<td>465</td>
<td>MACH</td>
<td>465</td>
<td>ENDM</td>
</tr>
<tr>
<td>466</td>
<td>MACH</td>
<td>466</td>
<td></td>
</tr>
<tr>
<td>467</td>
<td>MACH</td>
<td>467</td>
<td>INC MACRO DEST</td>
</tr>
<tr>
<td>468</td>
<td>MACH</td>
<td>468</td>
<td>INC MACRO INC, DEST</td>
</tr>
<tr>
<td>469</td>
<td>MACH</td>
<td>469</td>
<td>ENDM</td>
</tr>
<tr>
<td>470</td>
<td>MACH</td>
<td>470</td>
<td></td>
</tr>
<tr>
<td>471</td>
<td>MACH</td>
<td>471</td>
<td>DEC MACRO DEC, DEST</td>
</tr>
<tr>
<td>472</td>
<td>MACH</td>
<td>472</td>
<td>DEC MACRO DEC, DEST</td>
</tr>
<tr>
<td>473</td>
<td>MACH</td>
<td>473</td>
<td>ENDM</td>
</tr>
<tr>
<td>474</td>
<td>MACH</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>475</td>
<td>MACH</td>
<td>475</td>
<td>INC MACRO DEST, ADDR</td>
</tr>
<tr>
<td>476</td>
<td>MACH</td>
<td>476</td>
<td>INC MACRO DEC, ADDR</td>
</tr>
<tr>
<td>477</td>
<td>MACH</td>
<td>477</td>
<td>$SAVE GEN</td>
</tr>
<tr>
<td>478</td>
<td>MACH</td>
<td>478</td>
<td>JNZ ADDR</td>
</tr>
<tr>
<td>479</td>
<td>MACH</td>
<td>479</td>
<td>$RESTORE</td>
</tr>
<tr>
<td>480</td>
<td>MACH</td>
<td>480</td>
<td>ENDM</td>
</tr>
<tr>
<td>481</td>
<td>MACH</td>
<td>481</td>
<td></td>
</tr>
<tr>
<td>482</td>
<td>MACH</td>
<td>482</td>
<td>INC MACRO DEC, DEST</td>
</tr>
<tr>
<td>483</td>
<td>MACH</td>
<td>483</td>
<td>INC MACRO DEC, DEST</td>
</tr>
<tr>
<td>484</td>
<td>MACH</td>
<td>484</td>
<td>ENDM</td>
</tr>
<tr>
<td>485</td>
<td>MACH</td>
<td>485</td>
<td></td>
</tr>
<tr>
<td>486</td>
<td>MACH</td>
<td>486</td>
<td>INC MACRO INC, DEST</td>
</tr>
<tr>
<td>487</td>
<td>MACH</td>
<td>487</td>
<td>INC MACRO INC, DEST</td>
</tr>
<tr>
<td>488</td>
<td>MACH</td>
<td>488</td>
<td>ENDM</td>
</tr>
<tr>
<td>489</td>
<td>MACH</td>
<td>489</td>
<td></td>
</tr>
<tr>
<td>490</td>
<td>MACH</td>
<td>490</td>
<td>INC MACRO DEC, DEST</td>
</tr>
<tr>
<td>491</td>
<td>MACH</td>
<td>491</td>
<td>INC MACRO DEC, DEST</td>
</tr>
<tr>
<td>492</td>
<td>MACH</td>
<td>492</td>
<td>ENDM</td>
</tr>
<tr>
<td>493</td>
<td>MACH</td>
<td>493</td>
<td></td>
</tr>
<tr>
<td>494</td>
<td>MACH</td>
<td>494</td>
<td>INC MACRO INC, DEST</td>
</tr>
<tr>
<td>495</td>
<td>MACH</td>
<td>495</td>
<td>INC MACRO INC, DEST</td>
</tr>
<tr>
<td>496</td>
<td>MACH</td>
<td>496</td>
<td>ENDM</td>
</tr>
<tr>
<td>497</td>
<td>MACH</td>
<td>497</td>
<td></td>
</tr>
<tr>
<td>498</td>
<td>MACH</td>
<td>498</td>
<td>$EJECT</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ | LINE | SOURCE STATEMENT

503 : 504 : 505 : 506 : 507 : ALLOCATION OF MP I/O PORTS:
508 : 509 : 510 : 511 :
512 : 513 : BUS USED FOR DIRECT ADDRESS AND DATA TRANSFERS
514 : P1 USED AS INDIVIDUAL CONTROL OUTPUTS AND BREAK LOGIC
515 : P2 HIGH-ORDER ADDRESS AND ADDRESS SPACE SELECTION
516 :
517 : P0DIGIT EQU P7 USED TO ENABLE CHARACTERS AND STROBE ROWS OF KEYBOARD
518 : PSEGHI EQU P6 USED TO TURN ON III SEGMENTS OF CURRENTLY ENABLED DIGIT
519 : PSEGLO EQU P5 PORT FOR LOWER FOUR SEGMENTS
520 : PINPUN EQU P4 PORT USED TO SCAN FOR KEY CLOSURES
521 :
522 : INDIVIDUAL PINS OF PORT 1 USED AS FOLLOWS:
523 :
524 :
525 :
526 : ENDING OF PROGRAM EXECUTION
527 :
528 : ENDROM EQU 0000001B :P10 - HI ENABLES BREAK ON BREAK RAM OUTPUT SIGNAL
529 : ENBLNK EQU 0000010B :P11 - HI ENABLES BREAK ON RD OR WR TO LINK BY EP
530 : (NOTE: P11 & P10 BOTH HI ENABLES BREAK ON ANY EP INSTRUCTION CYCLE)
531 :
532 : EPSSTP EQU 0000100B :P12 - LO FORCES EP 55 INPUT LOW
533 : (HI GATES BREAKPOINT FLIP-FLOP TO EP 55 INPUT.
534 : CLRFIF EQU 0001000B :P13 - LO CLEARS BREAK FLIP-FLOP
535 :
536 : EPRESET EQU 0010000B :P14 - HI RESETS EP.
537 : MODOUT EQU 0100000B :P15 - LO WHEN EP IS EXECUTING USER PROGRAM.
538 :
539 : TOUT EQU 0100000B :P16 - SERIAL OUTPUT TO TTY OR CRT
540 :
541 :
542 $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE SOURCE STATEMENT

543:****************************************************************************
544:
545: INDIVIDUAL PINS OF PORT 2 USED AS FOLLOWS
546:
547:****************************************************************************
548:
549: PB3-P26: ADR11-00A0 FOR ACCESSING PROGRAM OR DATA RAM ARRAY
550:
0010 551 N0 LOU 000100008 :P24 - MEMORY MATRIX CONTROL PIN 0
0020 552 M1 EOU 001000008 :P25 - MEMORY MATRIX CONTROL PIN 1
0040 553 MUSEL EOU 010000008 :P26 - HIGH WHEN MP IN CONTROL OF COMMON MEM ARRAY,
554: LOW WHEN EP IN CONTROL
0080 555 EXPLEN EOU 100000008 :P27 - JUMPERED TO GROUND FOR STANDARD MONIT.
556: FLOATING WHEN EXPANSION MONITOR PRESENT
557:
558:
559: WHEN MP IN CONTROL OF MEMORY MATRIX M1-MB USED AS FOLLOWS:
560:
561: M1, MB MODE
562: 0 0 PROGRM RAM ARRAY ENABLED FOR READ & WRITE
563: 0 1 DATA RAM ARRAY ENABLED FOR READ & WRITE
564: 1 0 LINK REGISTER ENABLED FOR READ, RAM ARRAYS DISABLED.
565: (NOTE: LINK REGISTER ALWAYS ENABLED FOR MP WRITES)
566:
567: WHEN EP IN CONTROL OF MATRIX M1-MB USED AS FOLLOWS:
568:
569: M1, MB MODE
570: 0 0 EP PSN FETCHES FROM LINK REGISTER (USED TO FORCE OPCODES)
571: 1 0 EP PSN FETCHES FROM PROGRAM RAM ARRAY.
572: 1 1 EP RD & WR CONTROL DATA RAM ARRAY
573: 1 1 EP PSN FETCHES FROM PROGRAM RAM ARRAY.
574: RC & WR CONTROL LINK REGISTER.
575:
576 JFECT
LOC 0010

LINE   SOURCE STATEMENT

576. **********************
577. SYSTEM CONSTANT DEFINITIONS:
578. 
579. 580. -------------------------------
581. 
582. 583. -------------------------------
584. DECLARE CHANDI CONG... NUMBER OF DIGITS IN DISPLAY AND ROWS OF KEYS.
0008 585. CHANDI EQU 8
0009 586. 
0010 DECLARE NOLI CONG... LARGER DIMENSION OF KEYBOARD MATRIX
0014 587. NOLI EQU 4
0015 588. 
0016 DECLARE DECDE CONG... NUMBER OF SUCCESSIVE SCANS BEFORE KEY CLOSURE ACCEPTED
0018 589. DECDE EQU 8
0020 590. 
0021 DECLARE OYSE CONG... SIZE OF LARGEST MINI-MONITOR OVERLAY FOR EH:
0023 591. OYSE EQU 25
0025 592. 646. Declare Bufle CONG... LENGTH OF HEX FORMAT OMIT BUFFER (MAX RECORD LENGTH)
0029 593. Bufle EQU 16
0030 594. 
0031 DECLARE EROI CONG
0032 595. ZERO EQU 0
0033 596. DECLARE PLUSI CONG
0034 597. PLUSI EQU 1
0035 598. DECLARE PLUS2 CONG
0036 599. PLUS2 EQU 2
0037 600. DECLARE NEG1 CONG
0038 601. NEG1 EQU -1
0039 602. 
0040 DECLARE DETECT
0041 603. DETECT EQU 721
LOC 001   SOURCE STATEMENT

732 ;
733 ;*******************************************************************************
734 ;
735 ; BANK 0 REGISTER ALLOCATION.
736 ;
737 ;*******************************************************************************
738 ;
739 DECLARE DATA-R80   ; DATA USED BY LOGICAL ADDRESSING READ/WRITE UTILITIES
0002
740 DECLARE KEY-R80   ; HOLDS KEY CODE RETURNED FROM KBD INPUT ROUTINE
0003
741 DECLARE IMP-R80   ; COUNTER USED AS AN INDEX IN PARSER ROUTINE
0004
742 DECLARE CHKSUM-R80   ; CHECKSUM OF DATA BYTES TRANSMITTED IN HEX FILE FORMAT
0005
743 DECLARE DSTEMP-R80   ; TEMPORARY STORAGE FOR DISPLAY PATTERNS IN 'DSPCL'
0006
744 DECLARE XPCODE-R80   ; EXPANSION MONITOR ROUTINE CODE NUMBER
0007
745 DECLARE ROTPAT-R81   ; USED TO HOLD INPUT PATTERN BLIND ROTATED THROUGH CY
0008
746 DECLARE ROTPAT-R81   ; (ROTATION SET R2
0009
747 DECLARE ROTCNT-R81   ; COUNTS NUMBER OF BITS ROTATED THROUGH CY
0003
748 DECLARE ROTCNT-R81   ; (ROTATION SET R3
0004
749 DECLARE LASTXY-R81   ; HOLDS KEY POSITION OF LAST KEY DEPRESSION DETECTED
0004
750 DECLARE CURDIG-R81   ; HOLDS POSITION OF NEXT CHARACTER TO BE DISPLAYED
0005
751 DECLARE KEYFLG-R81   ; FLAG TO DETECT WHEN ALL KEYS ARE RELEASED
0006
752 DECLARE KEYFLG-R81   ; (REGISTER NOT USED FOR PRIMARY MONITOR)
0004
753 ;*******************************************************************************
754 ;
755 ;*******************************************************************************
756 ;
757 ;
758 ;
759 ;
760 ;
761 ;
762 ;
763 ;
764 ;
765 ;
766 ;
767 ;
768 ;
769 ;
770 ;
771 ;
772 ;
773 ;
774 ;
775 ;
776 ;
777 ;
778 ;
779 ;
780 ;
781 ;
782 ;
783 ;
784 ;
785 ;
786 ;
787 ;
788 ;
789 ;
790 ;
791 ;
792 ;
793 ;
794 ;
795 ;
796 ;
797 ;
798 ;
799 ;
800 ;
801 ;
802 ;
803 ;
804 ;
805 ;
806 ;
807 ;
808 ;
809 ;
810 ;
811 ;
812 ;
813 ;
814 ;
815 ;
816 ;
817 ;
818 ;
819 ;
820 ;
821 ;
822 ;
823 ;
824 ;
825 ;
826 ;
827 ;
828 ;
829 ;
830 ;
831 ;
832 ;
833 ;
834 ;
835 ;
836 ;
837 ;
838 ;
839 ;
840 ;
841 ;
842 ;
843 ;
844 ;
845 ;
846 ;
847 ;
848 ;
849 ;
850 ;
851 ;
852 ;
853 ;
854 ;
855 ;
856 ;
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
</table>
| 957 |  | ;
| 958 | | ;
| 959 |  | ;
| 960 | DATA RAM ALLOCATION | ;
| 961 |  | ;
| 962 |  | ;
| 963 |  | ;
| 964 | DECLARE EPPCC.RAM ; STORAGE IN MP FOR EP ACCUMULATOR | ;
| 965 | EPPCC EQU 32 | ;
| 966 | DECLARE EP.PSW.RAM ; STORAGE IN MP FOR EP PROGRAM STATUS WORD | ;
| 967 | EP.PSW EQU 32 | ;
| 968 | DECLARE EP.TIM.RAM ; STORAGE IN MP FOR EP TIMER/COUNT REGISTERS | ;
| 969 | EP.TIM.R EQU 34 | ;
| 970 | DECLARE EP.R0.RAM ; STORAGE IN MP FOR EP REGISTER 0 OF BANK 0 | ;
| 971 | EP.R0 EQU 35 | ;
| 972 | DECLARE EP.CLO.RAM ; STORAGE IN MP FOR LOW BYTE OF EP PROGRAM COUNTER | ;
| 973 | EP.CLO.EQU 36 | ;
| 974 | DECLARE EP.CHI.RAM ; STORAGE IN MP FOR HIGH NIBBLE OF EP PROGRAM COUNTER | ;
| 975 | EP.CHI.EQU 37 | ;
| 976 | DECLARE EPBLO.RAM ; PARAMETER 1 FOR SERIAL LINK DATA RATE GENERATOR | ;
| 977 | EPBLO.EQU 38 | ;
| 978 | DECLARE EPTHI.RAM ; PARAMETER 2 FOR SERIAL LINK DATA RATE GENERATOR | ;
| 979 | EPTHI.EQU 39 | ;
| 980 | DECLARE DSPTIM.RAM ; PARAMETER FOR AUTO-STEP AND AUTO-BREAK SEQUENCING RATE | ;
| 981 | DSPTIM.EQU 40 | ;
| 982 | DECLARE VERSNO.RAM ; MONITOR VERSION NUMBER | ;
| 983 | VERSNO.EQU 41 | ;
| 984 | DECLARE IREGA.RAM ; (UNUSED) | ;
| 985 | IREGA.EQU 42 | ;
| 986 | DECLARE IREGB.RAM ; (UNUSED) | ;
| 987 | IREGB.EQU 43 | ;
| 988 | DECLARE IREGC.RAM ; (UNUSED) | ;
| 989 | IREGC.EQU 44 | ;
| 990 | DECLARE IREGD.RAM ; (UNUSED) | ;
| 991 | IREGD.EQU 45 | ;
| 992 | DECLARE IREGE.RAM ; (UNUSED) | ;
| 993 | IREGE.EQU 46 | ;
| 994 | DECLARE IREGF.RAM ; (UNUSED) | ;
| 995 | IREGF.EQU 47 | ;
| 996 | DECLARE SMALO.RAM ; PRIMARY COMMAND STARTING MEMORY ADDRESS (LOW BYTE) | ;
| 997 | SMALO EQU 48 | ;
| 998 | DECLARE SMALHI.RAM ; PRIMARY COMMAND STARTING MEMORY ADDRESS (HIGH BYTE) | ;
| 999 | SMALHI EQU 49 | ;
| 1000 | DECLARE EMALO.RAM ; PRIMARY COMMAND ENDING MEMORY ADDRESS (LOW BYTE) | ;
| 1001 | EMALO EQU 50 | ;
| 1002 | DECLARE EMALHI.RAM ; PRIMARY COMMAND ENDING MEMORY ADDRESS (HIGH BYTE) | ;
| 1003 | EMALHI EQU 51 | ;
| 1004 | DECLARE MEMLO.RAM ; THIRD PARSER PARAMETER & HEX RECORD ADDRESS (LOW) | ;
| 1005 | MEMLO EQU 52 | ;
| 1006 | DECLARE MEMHI.RAM ; THIRD PARSER PARAMETER & HELX RECORD ADDRESS (HIGH) | ;
| 1007 | MEMHI EQU 53 | ;
| 1008 | DECLARE MEMC.RAM ; PRIMARY COMMAND NUMBER FROM PARSER TABLES (0-5) | ;
| 1009 | MEMC EQU 54 | ;
| 1010 | DECLARE TYPE.RAM ; PRIMARY COMMAND MODIFIER/OPTION (0-5) | ;
| 1011 | TYPE EQU 55 | ;

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0030</td>
<td></td>
<td>1100</td>
<td>DECLARE NUMCON.RAM ; MAX. NUMBER OF PARAMETERS ALLOWED FOR SELECTED COMMAND</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1105+</td>
<td>NUMCON EQU 56</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1109</td>
<td>DECLARE OPTION.RAM ; INDEX POINTER USED IN SEARCHING PARSER TABLES</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1104+</td>
<td>OPTION EQU 57</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1108</td>
<td>DECLARE NEXTPL.RAM ; CHARACTER POSITION FOR DISPLAY UTILITIES TO WRITE NEXT</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1203+</td>
<td>NEXTPL EQU 58</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1207</td>
<td>DECLARE KBDBUF.RAM ; POSITION OF KEY DEBOUNCED BY SCANNING SUBROUTINE</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1212+</td>
<td>KBDBUF EQU 59</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1216</td>
<td>DECLARE KEVLOC.RAM ; INCREMENTED AS SUCCESSIVE KEY LOCATIONS SCANNED</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1221+</td>
<td>KEVLOC EQU 60</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1225</td>
<td>DECLARE NEPTS.RAM ; KEEPS TRACK OF SUCCESSIVE RENDS OF SAME KEYSKROTE</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1230+</td>
<td>NEPTS LOU 61</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1234</td>
<td>DECLARE ASAVL.RAM ; HOLDS ACCUMULATOR VALUE DURING SERVICE ROUTINE</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1239+</td>
<td>ASAVL EQU 62</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1243</td>
<td>DECLARE RDelay.RAM ; COUNTER DECREMENTED WHEN AUTO-STEP DELAY IN PROGRESS</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1248+</td>
<td>RDelay EQU 63</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1252</td>
<td>DECLARE STRMP.RAM ; INDEX POINTER FOR DISPLAY CHARACTER STRING ACCESSING</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1257+</td>
<td>STRMP EQU 64</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1261</td>
<td>DECLARE BUFCON.RAM ; COUNT OF DATA BYTES IN HEX FORMAT RECORD BUFFER</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1266+</td>
<td>BUFCONT EQU 65</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1270</td>
<td>DECLARE RECTYP.RAM ; TYPE OF HEX FORMAT RECORD (0 OR 1)</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1275+</td>
<td>RECTYP EQU 66</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1279</td>
<td>DECLARE D.RAM ; BIT COUNTER FOR ASCII SERIAL I/O UTILITY SUBROUTINES</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1284+</td>
<td>D EQU 67</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1288</td>
<td>DECLARE REGC.RAM ; CHARACTER BEING SHIFTED DURING SERIAL I/O PROCESS</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1293+</td>
<td>REGC EQU 68</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1297</td>
<td>DECLARE H.RAM ; COUNTER IN SOFTWARE DELAY DATA RATE GENERATOR</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1302+</td>
<td>H EQU 69</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1306+</td>
<td></td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1307</td>
<td>DECLARE SEGMAP.CHRAND ; REGISTER ARRAY FOR DISPLAY PATTERNS</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1311+</td>
<td>SEGMAP EQU 70</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1314</td>
<td></td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1315</td>
<td>DECLARE OVSIZ; ; LOW-ORDER USER PROGRAM DURING MINI-MONITOR OVERLAYS</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1319+</td>
<td>OVSIZ; EQU 70</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1322</td>
<td></td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1323</td>
<td>DECLARE HEXBUF.DBUFLEN ; ALLOCATE BLOCK OF RAM FOR USE AS HEX RECORD BUFFER</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1327+</td>
<td>HEXBUF EQU 181</td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1330+</td>
<td></td>
</tr>
<tr>
<td>0030</td>
<td></td>
<td>1331</td>
<td>DECLARE RDEJT</td>
</tr>
</tbody>
</table>
LOC OBJ | LINE | SOURCE STATEMENT

0300 | 1332 | DATACLK 40
0300 | 1337+ | ORG 768
1341 | LINVLS TABLE OF CONSTANTS TO BE LOADED INTO MP INTERNAL RAM VARIABLES
1342 | AS PART OF SYSTEM INITIALIZATION PROCEDURE:
1343 |
1344 | **INITIAL VALUE** | **VARIABLE** | **TYPE**
1345 | ====== | ====== | ====== | ======
0000 00 | 1346 LINVLS: DB 00H :ROTIMT KB1
0001 00 | 1347 DB 00H :ROTCNT RB1
0002 00 | 1348 DB 00H :LSTKY RB1
0003 00 | 1349 DB CIARNO :CURDIG KB1
0004 00 | 1350 DB 00H :KEYFLG RB1
0005 00 | 1351 DB 00H :<REG7> RB1
0006 00 | 1352 DB 00H :LPACC RAM
0007 01 | 1353 DB 01H :LPPSW RAM
0008 00 | 1354 DB 00H :EPTIMR RAM
0009 00 | 1355 DU 00H :EPRO RAM
000A 00 | 1356 DB 00H :EPPOCLO RAM
000B 00 | 1357 DB 00H :EPPOCHI RAM
000C 03 | 1358 DB 03H :HBITLO RAM
000D 04 | 1359 DB 04H :HBITHI RAM
000E 03 | 1360 DB 03H :DSPTIM RAM
000F 35 | 1361 DU 35H :VERSND RAM
0110 00 | 1362 DB 00H :HREGA RAM
0111 00 | 1363 DB 00H :HREGB RAM
0112 00 | 1364 DB 00H :HREGC RAM
0113 00 | 1365 DB 00H :HREGD RAM
0114 00 | 1366 DB 00H :HREGE RAM
0115 00 | 1367 DB 00H :HREGF RAM
0116 00 | 1368 DB 00H :SMALO RAM
0117 00 | 1369 DB 00H :SMALHI RAM
0118 FF | 1370 DB 0FH :SNALO RAM
0119 0F | 1371 DB 0FH :SNALHI RAM
011A 00 | 1372 DB 00H :MEMLO RAM
011B 00 | 1373 DB 00H :MEMHI RAM
011C 00 | 1374 DB 00H :BCODE RAM
011D 04 | 1375 DB 04H :TYPE RAM
011E 01 | 1376 DB 01H :NUMCON RAM
011F 00 | 1377 DB 00H :OPTION RAM
0120 00 | 1378 DB CIARNO :NEXTPL RAM
0121 FF | 1379 DB 0FH :KDOBUF RAM
0122 00 | 1380 DB 00H :KEYLOC RAM
0223 | 1381 NOVALS EQU $-LINVLS
0223 | 1382 SIZE CHK
0223 | 1385+ SIZE SET 35
0223 | 1394:*****************************
0223 | 1396 $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE SOURCE STATEMENT

1397 $ INCLUDE (.FO:PARSER.MOD)
1398 CODEBLK 45
0800 1403 ORG 0
1407 ; INIT Initializes processor registers
1408 ; AND RAM locations 10 defined values.
0800 C5 =1409 INIT: SLL R00
0801 BF60 =1410 MOV XPCODE, #0
0803 74D1 =1411 CALL XPTEST
0805 27 =1412 CLR A
0806 3D =1413 MOVD PSEGLO,A
0807 3E =1414 MOVD PSEGH1, A
0808 B81A =1415 MOV R0,#1AH ; START AT RD1<REG2> = RAM LOC 1AH
0809 B923 =1416 MOV R1,# LOW NOVALS
080C B900 =1417 MOV R2,# LOW INVALS
080E FR =1418 INITLP: MOV A,R2
080F E3 =1419 MOVX3 A, R0
0810 A0 =1420 MOV @R0, A
0811 18 =1421 INC R0
0812 1A =1422 INC R2
0813 E9BE =1423 DJNZ KL, INITLP
0815 55 =1424 STRT 1
0816 74FF =1425 CALL EPSRK
0818 B88B =1426 MOV R0,#LOW(OVBAS+OVSIZE)
081A 74GA =1427 CALL OVLOAD
081C 54E5 =1428 CALL COMFil
081E D937 =1429 MOV R1,# TYPE
0820 11 =1430 INC OR1
0821 34F2 =1431 CALL INCMA
0823 54C5 =1432 CALL COMFil
0825 99FE =1433 ANL FL,#(NOT EPRESET) ; REMOVE EP RESET SIGNAL
0827 0429 =1434 JMP MAIN
0829 =1435 sizechk
082A =1436 SIZE SET 41
082B =1444
082C =1444
082D =1450 ~EJECT

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1451</td>
<td></td>
<td></td>
<td>KEYBOARD LAYOUT:</td>
</tr>
<tr>
<td>1452</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1453</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1454</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1455</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1456</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1457</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1458</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1459</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1460</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1461</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1462</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1463</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1464</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1465</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1466</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1467</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1468</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1469</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1470</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1471</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1472</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1473</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1474</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1475</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1476</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE SOURCE STATEMENT

-1477 ;
-1478 ;
-1479 ;
-1480 ;
-1481 ;
-1482 ;
-1483 ; KEY0 EQU 00H
-1484 ; KEY1 EQU 01H
-1485 ; KEY2 EQU 02H
-1486 ; KEY3 EQU 03H
-1487 ; KEY4 EQU 04H
-1488 ; KEY5 EQU 05H
-1489 ; KEY6 EQU 06H
-1490 ; KEY7 EQU 07H
-1491 ; KEY8 EQU 08H
-1492 ; KEY9 EQU 09H
-1493 ; KEYA EQU 0AH
-1494 ; KEYB EQU 0BH
-1495 ; KEYC EQU 0CH
-1496 ; KEYD EQU 0DH
-1497 ; KEYE EQU 0EH
-1498 ; KEYF EQU 0FH

0010 =1499 KEYTL EQU 10H
0012 =1500 KEYXT EQU 12H
0013 =1501 KEYEND EQU 13H
0014 =1502 KEYREL EQU 14H
0015 =1503 KEYPAT EQU 15H
0016 =1504 KEYDM EQU 16H
0017 =1505 KEYCLR EQU 17H
0018 =1506 KEYREC EQU 18H
0019 =1507 KEYRA EQU 19H
001A =1508 KEYPM EQU 1AH
001B =1509 KEYRG EQU 1BH
001C =1510 KEYLST EQU 1CH
001D =1511 KGORES EQU 1DH
001E =1512 KEYSO EQU 1EH
001F =1513 KEYMOD EQU 1FH
0086 =1514 KSETB EQU 68H
008C =1515 KCLR EQU 08H

0013 =1516 KBARY EQU 19H
0015 =1517 DBRK 19H
0011 =1520 RINT EQU 11H
0010 =1521 NDRK EQU 10H
0016 =1522 NDRK EQU 16H
001A =1523 SING EQU 1AH

-1524 ;
-1525 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC 00J  LINE  SOURCE STATEMENT

0829  =1526  CODEBLK 168
       =1531  ORG 41
       =1535  MAIN  OUTPUT_MESSAGE(COMMAND_PROMPT)
       =1536  CALL  INPUT_BYTE(KEY)
       =1537  MAIN2  IF THE KEY=END GO TO MAIN
       =1538  
       0829 BF01  =1539  MAIN:  MOV  XPCODE,.01
       082B 7401  =1540  CALL  XTTEST
       0820 2301  =1541  MOV  A,.01
       082F 3400  =1542  CALL  OUTUL
       0831 14EC  =1543  CALL  INKEY
       0833 FB    =1544  MAIN2:  MOV  A,KEY
       0834 D313  =1545  XRL  A,.KEYEND
       0836 C629  =1546  JZ  MAIN
       =1547  
       =1548  FINDOUT  IF THE KEY PRESSED IS A LEGITIMATE COMMAND INITIATOR:
       =1549  ;  LMP:=CTAB
       =1550  ;  BCODE:=TYPE:=0
       =1551  ;  WHILE  CTAB(LMP)<0  /CTAB EXHAUSTED/
       =1552  ;  IF CTAB(LMP)=KEY GO TO MAIN  /COMMAND ENTRY FOUND IN CTAB/
       =1553  ;  ELSE  LMP:=LMP+COMMAND_ENTRY_SIZE
       =1554  ;  BCODE:=BCODE+1
       =1555  
       =1556  ;  GOTO  ERROR
       0838 BC23  =1557  MOV  LMP,CTAB
       =1558  ;  MMOV  BCODE,ZERO
       0839 B036  =1559+  MOV  RL,#BCODE
       083C B100  =1560+  MOV  REL,ZERO
       083E B337  =1561+  MOV  RL,#TYPE
       0844 B100  =1562+  MOV  REL,ZERO
       0842 FC    =1563  FINDOUT:  MOV  A,LMP
       0843 E3    =1564  MOV#3  A,0A
       0844 B2BC  =1565  J55  MERROR
       0846 DB    =1566  XRL  A.KEY
       0847 C552  =1567  JZ  MAIN:
       0849 FC    =1568  MOV  A,LMP
       084A B383  =1569  ADD  A,#COMSIZ
       084C AC    =1570  MOV  LMP,A
       084D B936  =1571  MOV  RL,#BCODE
       084F 11    =1572  INC  REL
       0850 0442  =1600  JMP  FINDOUT
       =1681  
       =1682  ;  OUTPUT_MESSAGE(STROM(BCODE))  /*PROMPT FOR THE CURRENT COMMAND*/
       =1683  ;  I:=I+1
       =1684  ;  OPTION:=MEM(I)
       =1685  ;  I:=I+1
       =1686  ;  NO_OF_PARAMETERS:=MEM(I)
       =1687  ;  I:=3
       =1688  
       =1689  MAIN:  MMOV  A,BCODE
       0852 B936  =1619  MOV  RL,#BCODE
       0854 F1    =161A  MOV  A,REL
       0855 031D  =1623  ADD  A,#STROM
       0857 3482  =1624  CALL  OUTCLR

All mnemonics copyrighted © Intel Corporation 1976.

1-106
LOC  OBJ  LINE  SOURCE STATEMENT

0059 1C  =1625  INC  ITMP
005A  FC  =1626  MOV  R1,ITMP
005B  E3  =1627  MOVFP3  R,RA  ; GET OPTION POINTER
     =1628  MMov  OPTION.A
005C  8939  =1641+  MOV  R1,OPTION
005D  A1  =1642+  MOV  @R1.A
005E  1C  =1646  INC  ITMP
005F  FC  =1647  MOV  R1,ITMP
0061  E3  =1648  MOVFP3  R,RA  ; GET NO OF PARAMETERS
     =1649  MMov  NUMCON.A
0062  8938  =1652+  MOV  R1,NUMCON
0063  A1  =1653+  MOV  @R1.A
     =1657 ;  PARAMETER_BUFFER(0=255) =0
     =1659 ;
0065  8936  =1670  MOV  R1,#6  ; EACH PARAM IS 2 BYTES
0066  B938  =1671  MOV  @R0,#000H  ; START OF PARAM BUFFERS
0069  B900  =1672  MOV  @R0, R0H
006B  18  =1673  INC  R0
006C  9969  =1674  DJNZ  R1,MAINB
006E  146E  =1675  CALL  IMKEY
     =1676 ;  WHILE KEYS: OPTION+TYPE[6-8] DO
     =1677 ;  IF MENOPTION+TYPE[7]=1 GOTO MAIN1
     =1679 ;  TYPE:=TYPE+1
     =1680 ;  ENDWHILE
     =1681 ;
0070  B939  =1691  MOV  R1,OPTION
0072  F1  =1699+  MOV  @R1.A
0073  OC  =1712+  MOV  ITMP,A
0074  1C  =1715  INC  ITMP
     =1716  MAINC1  MMov  A,ITMP
0075  FC  =1732+  MOV  A,ITMP
0076  E3  =1736  MOVFP3  R,RA
0077  97  =1737  CLR  C
0078  F7  =1738  ELC  A
0079  77  =1739  RR  R  ; STRIP BIT SEVEN INTO CARRY
007A  D8  =1740  XRl  A,KEY
007B  0933  =1741  JZ  MAIND
007D  F687  =1742  JC  MAIND1
     =1743  MInc  TYPE
007F  B937  =1749+  MOV  R1,#TYPE
0081  F1  =1749+  MOV  A, R1.A
0082  17  =1753+  INC  A
0083  A1  =1758+  MOV  @R1.A
0084  1C  =1761  INC  ITMP
0085  8475  =1762  JMP  MAINC1
     =1763 ;  MODIFIER NOT FOUND SO RESET TYPE INDEX TO DEFAULT CASE (ZERO).
     =1764 ;
0087  B937  =1765  MOV  R1,#TYPE
0088  B100  =1776+  MOV  @R1,ZERO
     =1782  MMov  R,OPTION
LOC OBJ LINE SOURCE STATEMENT

0000 B939 =1791+ MOV RL:#OPTION
0000 F1 =1792+ MOV A:#R1
0000 E3 =1796 MOV A,R0
0000 3404 =1797 CALL OUTMSG
0001 849E =1798 JMP MAINB
=1799
=1800 ; CALL OUTPUT_MESSAGE(MODIFIER)
=1801 MAINB: MOV A:#OPTION
0003 B939 =1810+ MOV RL:#OPTION
0003 F1 =1811+ MOV A:#R1
0003 E3 =1815 MOV A,#R0
=1816 ADD A,#TYPE
0007 B937 =1822+ MOV RL:#TYPE
0009 61 =1823+ ADD A,#R1
0009 3404 =1825 CALL OUTMSG
000C 14EC =1828 CALL INFKEY
=1829
000E DC00 =1830 MAINB: MOV ITMP,#0
000A 2330 =1831 MAINB: MOV A,#SM1HO
000A GC =1832 ADD A,ITMP
000A GC =1833 ADD A,ITMP
000A 86 =1834 MOV B0,A
000A 14C9 =1835 CALL INFADDR
0007 F801 =1836 JC CMDINT
0009 1C =1837 INC ITMP
000A 8938 =1838 MOV RL:#NUMCON
000B 8939 =1839 MOV A,#R1
000B 07 =1840 DEC A
000A A1 =1841 MOV #R1,A
000A C68A =1842 JZ CMDINT
000B FB =1843 MOV A,#KEY
0002 D13 =1844 XRL A,#KEYEND
0004 C68A =1845 JZ CMDINT
0006 14EC =1846 CALL INFKEY
0008 849E =1847 JMP MAINB
=1848
=1849 ; CMDINT ENTER THE COMMAND PROCESSOR WITH:
=1850 ; BASE_CODE=THE MAIN COMMAND TYPE
=1851 ; TYPE-SUBCOMMAND TYPE
=1852 ; PARAMETER(1)=FIRST ADDRESS
=1853 ; PARAMETER(2)=SECOND ADDRESS
=1854 ; PARAMETER(3)=DATA
000A 4400 =1855 CMDINT: JMP IMPLEMENT
=1856
=1857 ; MERROR ERROR ENCOUNTERED IN MAIN PARSING ROUTINE.
000C D801 =1858 MERROR: MOV LDATA,#1
000E 249A =1859 JMP MERROR
=1860 SIZECHK
0007 =1863+ SIZE SET 151
=1864+
=1865+;*******************************************************************************
=1874 S EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC 001       LINE       SOURCE STATEMENT

0323     =1075       DATABLK 5B
0888+     =1088+     ORG 003
0884     =1084     
0885     =1085     ;******************************************************************************
0886     =1086     
0887     =1087     ; TABLES FOR PARSER
0888     =1088     
0889     =1089     ;******************************************************************************
0890     =1090     
0891     =1091     ; THE CIAB TABLE CONTAINS (COMSIZ) ENTRIES FOR EACH COMMAND. THE MEANING
0892     =1092     OF THE ENTRIES IS AS FOLLOWS:
0893     =1093     
0894     =1094     ; ENTRY 0 COMMAND KEY TO INITIATE
0895     =1095     ; ENTRY 1. POINTER TO THE LIST OF OPTIONS APPLICABLE TO THIS COMMAND
0896     =1096     ; ENTRY 2. NUMBER OF NUMERIC PARAMETERS REQUIRED BY THE COMMAND
0897     =1097     
0898     =1098     ;1888 CIAB EQU $ AND 0FFH
0899     =1099     COMSIZ EQU ?
0900     =1000     
0901     =1001     DB KEYMOD, LOW OPTAB1.1 ; EXAM
0902     =1002     DB KEYGR, LOW OPTAB3.1 ; GO
0903     =1003     DB KEYFL, LOW OPTAB1.3 ; FILL
0904     =1004     DB KEYLS, LOW OPTAB1.2 ; DUMP
0905     =1005     DB KEYRE, LOW OPTAB1.2 ; RECORD
0906     =1006     DB KEYRE, LOW OPTAB1.0 ; KELOAD
0907     =1007     DB KSETB, LOW OPTAB2.1 ; SETBRK
0908     =1008     DB KCLR, LOW OPTAB2.1 ; CLRBRK
0909     =1009     DB KGRES, LOW OPTAB3.0 ; GO FROM RESET STATE
0910     =1010     DB 0FFH ; ESCOP
0911     =1011     ;1912 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ   LINE   SOURCE STATEMENT

=1913 ;
=1914 ;   THE OPTION TABLE GIVES THE VARIOUS OPTIONS ALLOWED FOR EACH
=1915 ;   BASIC COMMAND, AS FOLLOWS:
=1916 ;
=1917 ;   ENTRY 0, START OF TABLE OF MODIFIER RESPONSES.
=1918 ;   ENTRY 1+, ALLOWED MODIFIER KEYSTROKES CORRESPONDING TO OPTIONS 0-5.
=1919 ;   NOTE THAT THE LAST BYTE IN EACH OPTION GROUP HAS BIT
=1920 ;   SEVEN SET TO INDICATE THE END.
=1921 ;
033F 2C  =1922 OPTAB1, DB  STRMEM
0340 10  =1923 DB  KEYP,KEYM,KEYREG,RINT
0341 16  =
0342 1B  =
0343 11  =
0344 12  =1924 DB  PBXK, DBXK OR 08H
0345 95  =
0346 2C  =1925 OPTAB2: DB  STRMEM
0347 1A  =1926 DB  KEYP,KEYM OR 08H
0348 96  =
0349 2C  =1927 OPTAB3, DB  STRGOC
034A 1B  =1928 DB  NOBK, WREN,SING
034B 16  =
034C 1A  =
034D 15  =1929 DB  KEYPAT,KEYTRA OR 08H
034E 99  =
             =1930 SIZECHK
002C  =1931 SIZE SET 44
      =1934:
      =1935:***************************************************************************
      =1944 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC   OBJ   LINE       SOURCE STATEMENT

0100   =1945  CODEBLK 130
        =1951  ORG 256
        =1959  OUTUTL OUTPUT ONE OF FOUR UTILITY DISPLAY PROMPTS (LEFT JUSTIFIED)
        =1968  ; ACCORDING TO ACC CONTENTS (0-3).
        =1961  OUTCLR CLEAR DISPLAY AND OUTPUT CHARACTER STRING STARTING
        =1962  ; AT THE ADDRESS POINTED TO BY BYTE AT ADDRESS IN ACCUMULATOR.
        =1963  OUTMSG SUBROUTINE TO COPY A STRING OF BIT PATTERNS FROM ROM TO THE
        =1964  ; DISPLAY REGISTERS.
        =1965  ; STRING SELECTED IS DETERMINED BY ACC WHEN CALLLD.
        =1966  ; ON ENTERING OUTMSG, ACC CONTENTS ARE USED TO ADDRESS A BYTE IN A
        =1967  ; LOOKUP TABLE ON THE CURRENT PAGE WHICH CONTAINS THE ADDRESS OF
        =1968  ; A STRING OF SEGMENT PATTERN DATA BYTES TO BE PRINTED UNTIL THE
        =1969  ; DISPLAY.
        =1971  ; CALLS SUBROUTINE 'HDISP'
        =1972  ; TO ACTUALLY EFFECT WRITING INTO THE DISPLAY REGISTERS.

0100  0319  =1973  OUTUTL: ADD A, #STRUL
0102  04F1  =1974  OUTCLR: CALL CLEAR
0104  08F3  =1975  OUTMSG: MOV R0, #STRMP
        =1976  ; MOV R1, #STRMP
0105  8940  =1983  ; MOV R0, #STRMP
0107  8941  =1990  ; MOV R1, #STRMP
0108  8940  =1997  ; MOV R0, #STRMP
0109  8A41  =1994  ; MOV R1, #STRMP
010A  8B41  =2001  ; MOV R0, #STRMP
010B  8C41  =2008  ; MOV R1, #STRMP
010C  8F41  =2015  ; MOV R0, #STRMP
010D  9048  =2022  ; MOV R1, #STRMP
010E  9148  =2029  ; JMP STRMP
010F  9248  =2036  ; JMP STRMP
0110  9348  =2043  ; STRMP EQU LOW $1
0111  9448  =2050  ; DB LOWN(DDATB)
0112  9548  =2057  ; DB LOWN(DDATB)
0113  9648  =2064  ; DB LOWN(DDATB)
0114  9748  =2071  ; DB LOWN(DDATB)
0115  A448  =2078  ; DB LOWN(DDATB)
0116  A548  =2085  ; DB LOWN(DDATB)
0117  A648  =2092  ; DB LOWN(DDATB)
0118  A748  =2099  ; DB LOWN(DDATB)
0119  A848  =2106  ; DB LOWN(DDATB)
011A  A948  =2113  ; DB LOWN(DDATB)
011B  AA48  =2120  ; DB LOWN(DDATB)
011C  AB48  =2127  ; DB LOWN(DDATB)
011D  AC48  =2134  ; DB LOWN(DDATB)
011E  AD48  =2141  ; DB LOWN(DDATB)
011F  AE48  =2148  ; DB LOWN(DDATB)
0120  AF48  =2155  ; DB LOWN(DDATB)
0121  B048  =2162  ; DB LOWN(DDATB)
0122  B148  =2169  ; DB LOWN(DDATB)
0123  B248  =2176  ; DB LOWN(DDATB)
0124  B348  =2183  ; DB LOWN(DDATB)
0125  B448  =2190  ; DB LOWN(DDATB)
0126  B548  =2197  ; DB LOWN(DDATB)
0127  B648  =2204  ; DB LOWN(DDATB)
0128  B748  =2211  ; DB LOWN(DDATB)

All mnemonics copyrighted © Intel Corporation 1976.

1-111
LOC OBJ LINE SOURCE STATEMENT

0129 69 =2051 DB LOW(DINTRO) ;DATA TYPE MODIFIER 3 RESPONSE ADDRESS
0129 65 =2052 DB LOW(DPRM) ;DATA TYPE MODIFIER 4 RESPONSE ADDRESS
0129 67 =2053 DB LOW(DDBR) ;DATA TYPE MODIFIER 5 RESPONSE ADDRESS
062C =2054 STRGOC EQU LOW $ ;
012C 88 =2055 DB LOW(DMORB) ;EXECUTION MODE MODIFIER 0
012D 60 =2056 DB LOW(DPRM) ;EXECUTION MODE MODIFIER 1
012C 6F =2057 DB LOW(DS5) ;EXECUTION MODE MODIFIER 2
012F 72 =2058 DB LOW(DFF) ;EXECUTION MODE MODIFIER 3
0130 75 =2059 DB LOW(DTR) ;EXECUTION MODE MODIFIER 4
0660 ;
0661 ; UTILITY OUTPUT MESSAGES
0662 ;
0131 79 =2064 DB 01111001B ;"E"
0132 50 =2065 DB 01101000B ;"R"
0133 50 =2066 DB 10100000B ;"R"
0134 5C =2067 DB 01111001B ;"D"
0135 50 =2068 DB 01101000B ;"R"
0136 C0 =2069 DB 11000000B ;"..."
0670 DSGNOM:
0137 80 =2071 DB 00000000B ; ""
0138 76 =2072 DB 01111101B ;"H"
0139 C0 =2073 DB 01101101B ;"S"
0139 79 =2074 DB 01111011B ;"E"
013A 40 =2075 DB 10000000B ;"...
013C 66 =2076 DB 01101101B ;"4"
013D E7 =2077 DB 11010111B ; ".9.(TM)"
013E 80 =2078 DRUN:
013F 80 =2079 DB 00000000B ; ""
0140 40 =2080 DB 01000000B ;"*"
0140 50 =2081 DB 01010000B ;"R"
0141 1C =2082 DB 00101000B ;"U"
0142 54 =2083 DB 01010100B ;"N"
0143 C0 =2084 DB 11000000B ; "."
0145 D7 =2085 DDPNT:
0144 73 =2086 DB 01110011B ;"F"
0145 E9 =2087 DB 10111001B ; ",C",
0146 30 =2088 $JECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OCJ   LINE   SOURCE STATEMENT

=2089 ;
=2090 ; PRIMARY COMMAND RESPONSE STRING PATTERNS
=2091 ;
=2092 DMOD:
0146 79 =2093 DB 01110010B.00110010B.11110100B  "ECH."
0147 39 =
0148 F4 =
=2094 DGO:
0149 3D =2095 DB 00111010B.11011100B  "GO."
014A DC =
=2096 DFILL:
014B 71 =2097 DB 01100001B.00110000B.10111000B  "FIL."
014C 30 =
014D 85 =
=2098 DLST:
014E 3B =2099 DB 00110000B.01101101B.11111000B  "LST."
014F 6D =
0150 F8 =
=2100 DREC:
0151 3E =2101 DB 00111100B.01100110B.10111000B  "UFL."
0152 73 =
0153 6C =
=2102 DREL:
0154 5E =2103 DB 01011100B.01010100B.10111000B  "DRL."
0155 54 =
0156 88 =
=2104 DSD:
0157 6D =2105 DB 01011010B.01111000B.11111000B  "STB."
0158 78 =
0159 FC =
=2106 DCD:
015A 39 =2107 DB 00111001B.00111000B.11111000B  "CLD."
015B 38 =
015C FC =
=2108 DGR:
015D 3D =2109 DB 00111010B.11010000B  "GR."
015E 08 =
=2110 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>=2111</td>
<td>=2112</td>
<td>MEMORY SPACE MODIFIER OPTION RESPONSE STRINGS</td>
<td></td>
</tr>
<tr>
<td>=2113</td>
<td>=2114</td>
<td>DPRMEM</td>
<td></td>
</tr>
<tr>
<td>0157 73</td>
<td>=2115</td>
<td>DB</td>
<td>011100110-11010000B</td>
</tr>
<tr>
<td>0160 DB</td>
<td></td>
<td></td>
<td>=2116</td>
</tr>
<tr>
<td>0161 SE</td>
<td>=2117</td>
<td>DB</td>
<td>010111100-11110111B</td>
</tr>
<tr>
<td>0162 F7</td>
<td>=</td>
<td></td>
<td>=2118</td>
</tr>
<tr>
<td>0163 50</td>
<td>=2119</td>
<td>DB</td>
<td>010100000-10111010B</td>
</tr>
<tr>
<td>0164 BD</td>
<td>=</td>
<td></td>
<td>=2120</td>
</tr>
<tr>
<td>0165 73</td>
<td>=2121</td>
<td>DB</td>
<td>011100110-11111100B</td>
</tr>
<tr>
<td>0166 FC</td>
<td>=</td>
<td></td>
<td>=2122</td>
</tr>
<tr>
<td>0167 SE</td>
<td>=2123</td>
<td>DB</td>
<td>010111100-11111100B</td>
</tr>
<tr>
<td>0168 FC</td>
<td>=</td>
<td></td>
<td>=2124</td>
</tr>
<tr>
<td>0169 76</td>
<td>=2125</td>
<td>DB</td>
<td>011101100-11010000B</td>
</tr>
<tr>
<td>016A DB</td>
<td>=</td>
<td></td>
<td>=2126</td>
</tr>
<tr>
<td>016B 54</td>
<td>=2127</td>
<td></td>
<td>RESPONSE MESSAGES FOR DU CONDITION MODIFIERS</td>
</tr>
<tr>
<td>016C FC</td>
<td>=</td>
<td></td>
<td>=2128</td>
</tr>
<tr>
<td>016D 7C</td>
<td>=2129</td>
<td>DB</td>
<td>010101000-11111100B</td>
</tr>
<tr>
<td>016E 0D</td>
<td>=</td>
<td></td>
<td>=2130</td>
</tr>
<tr>
<td>016F 6D</td>
<td>=2131</td>
<td>DB</td>
<td>011111000-11010000B</td>
</tr>
<tr>
<td>0170 6D</td>
<td>=</td>
<td></td>
<td>=2132</td>
</tr>
<tr>
<td>0171 FD</td>
<td>=</td>
<td></td>
<td>=2133</td>
</tr>
<tr>
<td>0172 77</td>
<td>=2134</td>
<td>DB</td>
<td>01101010-01101010-11111100B</td>
</tr>
<tr>
<td>0173 7C</td>
<td>=</td>
<td></td>
<td>=2135</td>
</tr>
<tr>
<td>0174 DB</td>
<td>=</td>
<td></td>
<td>=2136</td>
</tr>
<tr>
<td>0175 77</td>
<td>=2137</td>
<td>DB</td>
<td>011101110-01101010-11111100B</td>
</tr>
<tr>
<td>0176 6D</td>
<td>=</td>
<td></td>
<td>=2138</td>
</tr>
<tr>
<td>0177 FD</td>
<td>=</td>
<td></td>
<td>=2139</td>
</tr>
<tr>
<td>007F</td>
<td>=2140</td>
<td>SIZECHK</td>
<td></td>
</tr>
<tr>
<td>0879</td>
<td>=2143</td>
<td>SIZE SET 120</td>
<td></td>
</tr>
<tr>
<td>087A</td>
<td>=2144</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007F</td>
<td>=2145</td>
<td></td>
<td>1-114</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC 000  
LINE  SOURCE STATEMENT

00C0 =2155 CODEDLK 45
     =2160+ ORG 192
     =2164 ; INPADR INPUT DATA INTO TWO-BYTE PARAMETER BUFFER INDICATED BY R8.
     =2165 ; RECEIVE NUMERIC KEYS FROM KEYBOARD UNTIL ',', OR '.'.
     =2166 ; SHIFT INTO ADDRESS BUFFER.
     =2167 ; RE-WRITE DISPLAY.
     =2168 ; IF NUMBER OF CONSTANTS NEEDED IS ZERO, NO NEW PARAMETERS ARE ALLOWED.
     =2169 ;

00C0 97 =2170 INPADR: CLR  C
00C1 A7 =2171 CPL  C
     =2172 MMOV R8, NUMCON
00C2 6D38 =2181+ MOV R1, INUMCON
00C4 F1 =2182+ MOV R8, R1
00C5 6D7 =2186 JZ ELSIF1
00C7 FB =2187 INPAD1: MOV R1, KEY
00C8 9207 =2188 JB4 ELSIF1
00C9 20 =2189 XCH R8, R0
00CB 47 =2190 SWAP R
00CC 20 =2191 XCH R8, R0
00CD 30 =2192 XCHD R8, R0
00CE 18 =2193 INC R8
00CF 30 =2194 XCHD R8, R0
00D0 3478 =2195 CALL UPDATE
00D2 14EC =2196 CALL INFKEY
00D4 97 =2197 CLR C
00D5 84C7 =2198 JMP INPAD1
     =2199 ;
     =2200 ; ELSIF1 IF KEY= ',', OR '.' THEN RETURN.
     =2201 ;
00D7 FB =2202 LLSIF1: MOV R1, KEY
00D8 D312 =2203 XRL R1, #KEYEXT
00DA 6E5 =2204 JZ ELSIF2
00C8 FB =2205 MOV R1, KEY
00DD 3D13 =2206 XRL R1, #KEYEND
00DF 6E5 =2207 JZ ELSIF2
     =2208 ;
     =2209 ; ELSE GOTO PERROR.
     =2210 ;
00E1 B802 =2211 MOV LDATA #2
00E2 249A =2212 JMP PERROR
00E5 B846 =2213 ELSIF2: MOV R8, #SEGMAP
00E7 B803 =2214 MOV R1, R3
00E9 0AF5 =2215 CALL DLINK
00ED 63 =2216 RET
     =2217 SIZECHK
00EC =2220+ SIZE SET 44
     =2221+;
     =2222+;******************************************************************************
     =2231 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE SOURCE STATEMENT
=2232 CODEBK 35
=2242+ ORG 376
=2246:UPDATE UPDATE ADDRESS FIELD
=2247; (LAST THREE CHARACTERS OF DISPLAY) WITH ADDRESS BUFFER
=2248 UPDATE: MOV NEXTPL, PLUS3
=2259+ MOV RL #EXTPL
=2260+ MOV ORL #PLUS3
=2264; WRITE ADDR INTO NEXT THREE BUFFER LOCATIONS.
=2265 UPDATE: MOV A, #000
=2266 DEC R0
=2267 ANL A, #08H
=2268 JNZ DSPHI
=2269 CALL WDISP
=2270 MOV A, #000
=2271 SWAP A
=2272 ANL A, #08H
=2273 JNZ DSPM1
=2274 CALL WDISP
=2275 JMP DSPLO
=2276 DSPHI: CALL DSPACC
=2277 DSPM1: MOV A, #000
=2278 SWAP A
=2279 DSPM1: CALL DSPACC
=2280 DSPLO: MOV A, #000
=2281 CALL DSPACC
=2282 RET
=2283 SIZECHK
=2284+ SIZE SET '32
=2285;*******************************************************************************
=2286+ EJECT
LOC OBJ        LINE  SOURCE STATEMENT

=2208  CODEBLK 35
=2308+  ORG  486
=2312  ;ERROR.  REPEAT
=2313 ;  OUTPUT_MESSAGE([ERROR]_PROMPT)
=2314 ;  OUTPUT([DATA])
=2315 ;  CALL INPUT_BYTE([KEY])
=2316 ;  UNTIL KEY="CLEAR" / "PREVIOUS"

=2317  ERROR:  MOV  [DATA] #4
=2319  ERROR:  MOV  [XCODE] #2
=231A  CALL  XPTTEST
=231E  CLR  A
=231F  MOV  P97.W.A
=2320  MOV  A. KEY
=2321  XRL  A. #KEYCLR
=2322  JZ  ERROR2
=2323  CLR  A
=2324  CALL  OUTPUT
=2325  MOV  A. [DATA]
=2326  CALL  DSPACE
=2327  MOV  KEUDF. NEG1
=2328  MOV  R1. #DDOF
=2329  MOV  R3. #NEG1
=232A  CALL  INKEY
=232B  MOV  R1. #NEG1
=232C  JNZ  ERROR2
=232D  CLR  A
=232E  CALL  OUTPUT
=232F  MOV  R1. #RR
=2330  CALL  JFIRST
=2331  MOV  R1. #RR
=2332  JNZ  RERROR
=2333  ERROR:  JMP  MAIN
=2334  SIZE CLR
=2335+  SIZE SET 32
=2336 ;
=2337  CODEBLK 80
=2338+  ORG  512
=2339  ;IMPLEM IMPLEMENT COMMAND
=233A  IMPLEM:  MOV  A. [LOW] (JMTBL)
=233B  MOVO  A. [BUCOC]
=233C  MOV  R1. #BCODC
=233D  MOV  R1. #CODEC
=233E  MOV  R1. #RR
=233F  MOV  A. [RR]
=2340 ;
=2341  JMP  JMTBL:
=2342  ;
=2343  DB  LOW(JTOMOD)
=2344  DB  LOW(JTOSL)
=2345  DB  LOW(JTOFIL)
=2346  DB  LOW(JTOLST)
=2347  DB  LOW(JTOREC)
=2348  DB  LOW(JTOREL)
=2349  DB  LOW(COMSQR)
=234A  DB  LOW(JTOKES)
=234B  DB  LOW(JGORES)
=234C  DB  LOW(JTOMOD)
=234D  JMP  EXAMIN
=234E ;
=234F  DB  STD:  HEX FORMAT DATA DUMP

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE SOURCE STATEMENT

0212 B472 =2413 CALL HFILEO
0214 B429 =2414 JMP MAIN
=2415 ;
0216 5497 =2416 JIOREL CALL RECEIN
0218 0429 =2417 JMP MAIN
=2418 ;
021A 85 =2419 JTOLST: CLR F0
021B 95 =2420 CPL F0
021C B472 =2421 CALL HFILEO
021E 0429 =2422 JMP MAIN
=2423 ;
0220 0408 =2424 JTOGO: JMP EPRUN
=2425 ;
0222 54E5 =2426 JIOFIL: CALL COMFIL
0224 0429 =2427 JMP MAIN
=2428 ;
0226 0461 =2429 JGORES: JMP COMGDR
=2430 ;
=2431 :COMSR: COMMAND TO CLEAR BREAKPOINTS
0228 0908 =2432 COMSR: MOV LDTA. #:0
022D 442E =2433 JMP BRXFI
=2434 ;
=2435 :COMSR: COMMAND TO SET BREAKPOINTS
022C B001 =2436 COMSR: MOV LDTA. #:1
022E 2304 =2437 BRXFI: MOV A. #:4
=2438 MADD TYPB. A
0230 0327 =2448 MOV RL. #:TYPE
0232 61 =2449 ADD A. #:1
0233 F1 =2455 MOV MDLA
0234 F400 =2459 BRKND: CALL LSTORE
0236 F8 =2460 MOV A. KEV
0237 0313 =2461 XRL A. BKEYEND
0239 C640 =2462 JE BRKND
023B 14EC =2463 CALL INPEKEY
=2464 MMOV NUMCON. PLUSI
023D 0330 =2475 MOV RL. #:NUMCON
023F 0101 =2476 MOV ARL. #:PLUSI
0241 B030 =2488 MOV RO. #:MAPLO
0243 B000 =2481 MOV #O. #:8
=2482 MMOK SMWAI. ZERO
0245 D931 =2493 MOV RL. #:SMWAI
0247 B100 =2494 MOV ARL. #:ZERO
0249 14C0 =2498 CALL INPADDR
024B E634 =2499 JNC BRKND
024D 0429 =2500 BRKND: JMP MAIN
=2501 SIZEDIK
024F =2504 SIZE SET 79
=2505+;
=2506+;******************************************************************************
=2513 $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC  OBJ  LINE  SOURCE STATEMENT

024  =2516  C0DECBLK 75
     =2531  ORG  591
     =2535  EXAMIN EXAMINE/MODIFY MEMORY COMMAND
     =2536  ; DISPLAYS MEMORY ADDRESS SPACE OPTION, ADDRESS VALUE, AND CURRENT DATA.
     =2537  ; READS KEYBOARD AND INTERPRETS RESPONSE.
     =2538  ;
     =2539  ; OUTPUT MESSAGE("MEMORY_SOMPE_OPTION=<DATA_BTYE>)

024F  =2540  EXAMIN:  CLR  F0
     =2541  EXAMIN:  ANO  A1_TYPE
0250  0337  =2546  MOV  R1_Type
0252  F1  =2551  MOV  A0_R1
0253  0326  =2555  ADD  #STRAMEM :OFFSET FOR FIRST MEMORY TYPE STRING
0255  3402  =2556  CALL  OUTCLR
0257  031  =2557  MOV  R0.#SMAD0:1
0259  347C  =2558  CALL  UAQAD0
025B  2848  =2559  MOV  A. #1B0D1880'B : ;
025D  0408  =2560  CALL  ADISP
025F  14FC  =2561  CALL  LFETCH
0261  F9  =2562  MOV  A1.LDATA
0262  47  =2563  SWAP  A
0263  0403  =2564  CALL  DSPACC
0265  FA  =2565  MOV  A.1.LDATA
0266  0403  =2566  CALL  DSPACC
     =2567  ;
     =2568  ;
     =2569  ; INPUT.KEY(KEY)
     =2570  ; IF (KEY IS NOT NUMERIC)
     =2571  ; IF (KEY=KEYEND) GO TO PARSER
     =2572  ; ELSEIF (KEY=KEYNEXT)
     =2573  ; INCREMENT <SMAD>
     =2574  ; GOTO EXAMIN
     =2575  ; ELSEIF (KEY=KEYPREVIOUS)
     =2576  ; DECREMENT <SMAD>
     =2577  ; GOTO EXAMIN
     =2578  ; ELSE GOTO PERR0
     =2579  ;
026C  14EC  =2580  CALL  INPKEY
     =2581  MMOV  A.KEY
026F  FB  =2597  MOV  A.KEY
026B  9278  =2601  JB4  EXAMIN
     =2602  ; APPEND DATA WITH "L Willie (KEY)"
     =2603  ; CALL LISTORE
     =2604  ; APPEND DATA WITH "LWillie_OKY"
     =2605  ; GOTO EXAMIN
     =2606  ;
026D  FA  =2607  MOV  A1.LDATA
026E  47  =2608  SWAP  A
026F  53F0  =2609  ANL  A. #0FH
0271  D675  =2610  JFB  EXAMS
0273  27  =2611  CLR  A
0274  95  =2612  CPL  F0
0275  68  =2613  EXAMS:  ADD  A.KEY
0276  FA  =2614  MOV  LDATA, A
0277  F400  =2615  CALL  LISTORE
0279  4450  =2616  JMP  EXAMIN

All mnemonics copyrighted © Intel Corporation 1976.

1-119
LOC  OBJ  LINE  SOURCE STATEMENT

=2617 ;
0270 D313 =2618 EXAM1: XRL A, #(KEYEND)
027D 9681 =2619 JNZ EXAM2
027F 8429 =2620 JMP MAIN
=2621 ;
0281 FD =2622 EXAM2: MOV A, KEY
0282 D312 =2623 XRL A, #KEYNEXT
0284 968A =2624 JNZ EXAM3
0286 34F2 =2625 CALL INCMA
0288 444F =2626 JMP EXAMIN
028A FD =2627 EXAM3: MOV A, KEY
028B D317 =2628 XRL A, #KEYCLR
028D 9683 =2629 JNZ EXAM4
028F 54F4 =2630 CALL DECMA
0291 444F =2631 JMP EXAMIN
0293 B003 =2632 EXAM4: MOV LDATA, #03H
0295 249A =2633 JMP PERROR
=2634 SIZECHK
0048 =2637+ SIZE SET 72
=2638+
=2639+;******************************
=2640 ;
=2649  CODEBLK 4
00EC =2654+ OKG 236
00EC D4C2 =2658 IMPLY: CALL KEKIN ; RETURNS KEY DEPRESSION IN A
00EE AD =2659 MOV KEV, A
00EF 83 =2660 RET
=2661 SIZECHK
0004 =2664+ SIZE SET 4
=2665+
=2666+;******************************
=2675 $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC  OBJ  LINE  SOURCE STATEMENT

2676 $  INCLUDE( FB: GDCONS.MOD)
=2677  CODEBK 210
0400  =2697+  ORG  1824
=2701  ;EPRUN  RUN EMULATION MODE.
=2702  ;RELDO  EP WITH SYSTEM STATUS AND RELEASE.
=2703  ;SEQUENCE IS AS FOLLOWS:
=2704  ;IF COMMAND WAS TERMINATED BY THE 'NEXT' KEY:
=2705  ;STORE SNA INTO EP PC;
=2706  ;STORE EP PC INTO TOP-OF-STACK (RELATIVE TO EP SW);
=2707  ;PASS EP RA;
=2708  ;PASS EP PSW;
=2709  ;PASS EP TIMER;
=2710  ;PASS EP ACCUMULATOR;
=2711  ;
0400  2382  =2712  EPRUN:  MOV  A, #2
0402  3480  =2713  CALL  OUTUTL
=2714  MOV  A, NUMCON
0404 B938  =2723+  MOV  RL, NUMCON
0406 F1  =2724+  MOV  A, ARL
0407 9615  =2725+  MOV  JLZ  EPCONT
=2726  MOV  EPPCL0, SNAI0
0409 B930  =2727+  MOV  RL, ISMAI0
040B F1  =2728+  MOV  A, ARL
040C B924  =2729+  MOV  RL, REPCLO
040E R1  =2730+  MOV  ARL, A
=2731  MOV  EPPCHI, SNAII
040F B931  =2732+  MOV  RL, ISNAII
0411 F1  =2733+  MOV  A, ARL
0412 B925  =2734+  MOV  RL, REPCII
0414 R1  =2735+  MOV  ARL, A
=2736  MOV  EPCII4, EPCII5
=2737  MOV  EPRCII, SIIII
=2738  MOV  EPRCII, 5IflO,
=2739  MOV  EPRCII, RII,
=2740+  MOV  A, EPPSW
041C B921  =2800+  MOV  RL, REPPSW
041E R1  =2801+  MOV  ARL, A
=2802  EPCON1:  MOV  A, KEY
=2803  XRL  A, KEY NXT
0418 061F  =2805  JMZ  EPCON1
041A 2301  =2786  MOV  A, #10H ;STACK ONE LEVEL DEEP TO HOLD USER STARTING ADDRESS
=2787  MOV  EPPSW, A
041C B921  =2808+  MOV  RL, REPPSW
041E R1  =2809+  MOV  ARL, A
=2810  EPCON1:  MMov  DLAT, EPPCL0
041F B924  =2821+  MOV  RL, REPCLO
0421 F1  =2822+  MOV  A, ARL
0422 AA  =2835+  MOV  DLAT, A
=2836  MMov  A, EPPSW
0423 B921  =2847+  MOV  RL, REPPSW
0425 F1  =2848+  MOV  A, ARL
=2852  DEC  A
0426 5307  =2853  ANL  A, #07H
0429 E7  =2854  RL  A
042A 0308  =2855  ADD  A, #07H
=2856  MMov  SNAII, A
042C B930  =2869+  MOV  RL, ISMAI0
042E R1  =2870+  MOV  ARL, A
042F F4C3  =2874  CALL  EPCSTOR
=2875  MINC  SNAII
0431 B930  =2880+  MOV  RL, ISMAII0
0433 F1  =2881+  MOV  A, ARL

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OFFSET</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0434</td>
<td>17</td>
<td>=2885+</td>
<td>INC  A</td>
</tr>
<tr>
<td>0435</td>
<td>01</td>
<td>=2888+</td>
<td>MOV  ARL.A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=2893</td>
<td>MMov  A,EPPSW</td>
</tr>
<tr>
<td>0436</td>
<td>8921</td>
<td>=2892+</td>
<td>MOV  RL,EPPSW</td>
</tr>
<tr>
<td>0438</td>
<td>F1</td>
<td>=2893+</td>
<td>MOV  A. ARL.</td>
</tr>
<tr>
<td>0439</td>
<td>53F0</td>
<td>=2897</td>
<td>ANL  A. RA.0FAH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=2898</td>
<td>MORL  A. EPPCHI</td>
</tr>
<tr>
<td>043B</td>
<td>8925</td>
<td>=2898+</td>
<td>MOV  RL.EPPCHI</td>
</tr>
<tr>
<td>043D</td>
<td>41</td>
<td>=2904+</td>
<td>ORL  A. ARL.</td>
</tr>
<tr>
<td>043E</td>
<td>08</td>
<td>=2909</td>
<td>MOV  LDAM. A</td>
</tr>
<tr>
<td>043F</td>
<td>F403</td>
<td>=2910</td>
<td>CALL  EPSTOR</td>
</tr>
<tr>
<td>0441</td>
<td>8931</td>
<td>=2911+</td>
<td>MOV  RL. BLOW(0V2BS+0V512)</td>
</tr>
<tr>
<td>0443</td>
<td>746A</td>
<td>=2912</td>
<td>CALL  OVLAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=2913</td>
<td>MMov  A. EMM.0</td>
</tr>
<tr>
<td>0445</td>
<td>8923</td>
<td>=2914+</td>
<td>MOV  RL. NEPR8</td>
</tr>
<tr>
<td>0447</td>
<td>F1</td>
<td>=2915+</td>
<td>MOV  A. ARL.</td>
</tr>
<tr>
<td>0448</td>
<td>F400</td>
<td>=2916+</td>
<td>CALL  EPPASS</td>
</tr>
<tr>
<td>044A</td>
<td>8921</td>
<td>=2917+</td>
<td>MOV  RL. EPPSW</td>
</tr>
<tr>
<td>044C</td>
<td>F1</td>
<td>=2918+</td>
<td>MOV  A. ARL.</td>
</tr>
<tr>
<td>044D</td>
<td>F400</td>
<td>=2919+</td>
<td>CALL  EPPASS</td>
</tr>
<tr>
<td>044F</td>
<td>8922</td>
<td>=2920+</td>
<td>MOV  RL. NEPTIMR</td>
</tr>
<tr>
<td>0451</td>
<td>F1</td>
<td>=2921+</td>
<td>MOV  A. ARL.</td>
</tr>
<tr>
<td>0452</td>
<td>F400</td>
<td>=2922+</td>
<td>CALL  EPPASS</td>
</tr>
<tr>
<td>0454</td>
<td>8920</td>
<td>=2923+</td>
<td>MOV  RL. NEPACC</td>
</tr>
<tr>
<td>0456</td>
<td>F1</td>
<td>=2924+</td>
<td>MOV  A. ARL.</td>
</tr>
<tr>
<td>0457</td>
<td>F400</td>
<td>=2925+</td>
<td>CALL  EPPASS</td>
</tr>
<tr>
<td>0459</td>
<td>8903</td>
<td>=2926+</td>
<td>ORL  P1. 0000011B</td>
</tr>
<tr>
<td>045B</td>
<td>F406</td>
<td>=2927+</td>
<td>CALL  EPSTEP</td>
</tr>
<tr>
<td>045D</td>
<td>745A</td>
<td>=2928+</td>
<td>CALL  OVSMP</td>
</tr>
<tr>
<td>045F</td>
<td>846B</td>
<td>=2929+</td>
<td>JMP  CGO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=2930+</td>
<td>;CONGOR  GO FROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RESET COMMAND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=2931+</td>
<td>;RESET PROCESSOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=2932+</td>
<td>;RELOAD LOW ORDER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PROGRAM BYTES INTO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PROGRAM MEMORY</td>
</tr>
<tr>
<td>0461</td>
<td>2382</td>
<td>=2933+</td>
<td>MOV  A. A.2</td>
</tr>
<tr>
<td>0463</td>
<td>3408</td>
<td>=2934+</td>
<td>CALL  OUTUTL</td>
</tr>
<tr>
<td>0465</td>
<td>8918</td>
<td>=2935+</td>
<td>ORL  P1. #EPRSE1</td>
</tr>
<tr>
<td>0467</td>
<td>745A</td>
<td>=2936+</td>
<td>CALL  OVSMP</td>
</tr>
<tr>
<td>0469</td>
<td>99EF</td>
<td>=2937+</td>
<td>ANL  P1. #1(NOT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>EPRSET)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=2938+</td>
<td>;CGO  SET UP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BREAK LOGIC FOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>APPROPRIATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BREAK CONDITIONS;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>=3000+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3001+</td>
<td>;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3002+</td>
<td>;CGO.  MMov  A.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TYPE</td>
</tr>
<tr>
<td>046C</td>
<td>8937</td>
<td>=3003+</td>
<td>MOV  RL. #TYPE</td>
</tr>
<tr>
<td>046F</td>
<td>F1</td>
<td>=3004+</td>
<td>MOV  A. ARL.</td>
</tr>
<tr>
<td>046E</td>
<td>0371</td>
<td>=3005+</td>
<td>ADD  A. ALow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>GOTBL</td>
</tr>
<tr>
<td>0470</td>
<td>D3</td>
<td>=3006+</td>
<td>JMP  #A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3007+</td>
<td>;</td>
</tr>
<tr>
<td>0471</td>
<td>7C</td>
<td>=3008+</td>
<td>1GOTBL:  DE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LOW(CONE)</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0472</td>
<td>76</td>
<td>=3828</td>
<td>DB LOW(CG0WD)</td>
</tr>
<tr>
<td>0473</td>
<td>80</td>
<td>=3821</td>
<td>DB LOW(CG0SS)</td>
</tr>
<tr>
<td>0474</td>
<td>76</td>
<td>=3822</td>
<td>DB LOW(CG0PAT)</td>
</tr>
<tr>
<td>0475</td>
<td>80</td>
<td>=3823</td>
<td>DB LOW(CG0TRA)</td>
</tr>
<tr>
<td>=3824</td>
<td></td>
<td>=3825</td>
<td>CG0PAT:</td>
</tr>
<tr>
<td>0476</td>
<td>99FD</td>
<td>=3826</td>
<td>CG0WD ANL P1, NOT 00000010B</td>
</tr>
<tr>
<td>0477</td>
<td>8501</td>
<td>=3827</td>
<td>ORL P1, 000000001B</td>
</tr>
<tr>
<td>0478</td>
<td>8482</td>
<td>=3828</td>
<td>JMP EPRUN4</td>
</tr>
<tr>
<td>=3829</td>
<td></td>
<td>=3830</td>
<td>CG0ND ANL P1, NOT 00000011B</td>
</tr>
<tr>
<td>0479</td>
<td>8482</td>
<td>=3831</td>
<td>JMP EPRUN4</td>
</tr>
<tr>
<td>=3832</td>
<td></td>
<td>=3833</td>
<td>CG0TRA:</td>
</tr>
<tr>
<td>0480</td>
<td>8983</td>
<td>=3834</td>
<td>ORL P1, 00000011B</td>
</tr>
<tr>
<td>=3835</td>
<td></td>
<td>=3836</td>
<td>EPRUN4 SET UP CONTROL LOGIC TO RUN USER'S PROGRAM.</td>
</tr>
<tr>
<td>=3837</td>
<td></td>
<td>=3838</td>
<td>RELEASE PROCESSOR TO RUN.</td>
</tr>
<tr>
<td>0482</td>
<td>8R20</td>
<td>=3839</td>
<td>ORL P2, 00010000B</td>
</tr>
<tr>
<td>0484</td>
<td>9HFF</td>
<td>=3840</td>
<td>ANL P2, NOT 00010000B</td>
</tr>
<tr>
<td>0486</td>
<td>99DF</td>
<td>=3841</td>
<td>ANL P1, NOT MODOUT</td>
</tr>
<tr>
<td>0488</td>
<td>F4F4</td>
<td>=3842</td>
<td>CALL EPRUN1</td>
</tr>
<tr>
<td>=3843</td>
<td></td>
<td>=3844</td>
<td>WAIT FOR KEYSTROKE INPUT OR HARDWARE BREAK TO OCCUR.</td>
</tr>
<tr>
<td>=3845</td>
<td></td>
<td>=3846</td>
<td>EPRUN1: CALL TOPPOL</td>
</tr>
<tr>
<td>048C</td>
<td>F4AF</td>
<td>=3847</td>
<td>CALL KBDPOL</td>
</tr>
<tr>
<td>048E</td>
<td>37</td>
<td>=3848</td>
<td>CPL A</td>
</tr>
<tr>
<td>048F</td>
<td>6295</td>
<td>=3849</td>
<td>JB7 EPRUN3</td>
</tr>
<tr>
<td>0491</td>
<td>8639</td>
<td>=3850</td>
<td>JNI EPRUN2</td>
</tr>
<tr>
<td>0493</td>
<td>848A</td>
<td>=3851</td>
<td>JMP EPRUN1</td>
</tr>
<tr>
<td>=3852</td>
<td></td>
<td>=3853</td>
<td>EPRUN2 A KEYSTROKE WAS DETECTED WHILE EP WAS RUNNING.</td>
</tr>
<tr>
<td>=3854</td>
<td></td>
<td>=3855</td>
<td>BREAK EXECUTION.</td>
</tr>
<tr>
<td>=3856</td>
<td></td>
<td>=3857</td>
<td>PROCESS KEYSTROKE.</td>
</tr>
<tr>
<td>0496</td>
<td>D480</td>
<td>=3858</td>
<td>EPRUN3: CALL STSAVE</td>
</tr>
<tr>
<td>0497</td>
<td>8483</td>
<td>=3859</td>
<td>JMP EPRUN5</td>
</tr>
<tr>
<td>=3860</td>
<td></td>
<td>=3861</td>
<td>EPRUN4: CALL STSAVE</td>
</tr>
<tr>
<td>=3862</td>
<td></td>
<td>=3863</td>
<td>MMOV A, TYPE</td>
</tr>
<tr>
<td>0499</td>
<td>B3F7</td>
<td>=3864</td>
<td>MOV R1, #TYPE</td>
</tr>
<tr>
<td>049D</td>
<td>F1</td>
<td>=3865</td>
<td>MOV R0, R0</td>
</tr>
<tr>
<td>049E</td>
<td>0341</td>
<td>=3866</td>
<td>ADD A, LOW CNTTAB</td>
</tr>
<tr>
<td>04AF</td>
<td>03</td>
<td>=3867</td>
<td>JMP &amp;A</td>
</tr>
<tr>
<td>=3868</td>
<td></td>
<td>=3869</td>
<td>MOV R0, A</td>
</tr>
<tr>
<td>04A1</td>
<td>A6</td>
<td>=3870</td>
<td>MOV R0, LOW(BKERR)</td>
</tr>
<tr>
<td>04A2</td>
<td>DB</td>
<td>=3871</td>
<td>MOV R0, LOW(EPRUN)</td>
</tr>
<tr>
<td>04A3</td>
<td>DB</td>
<td>=3872</td>
<td>MOV R0, LOW(EPRUN)</td>
</tr>
<tr>
<td>04A4</td>
<td>AN</td>
<td>=3873</td>
<td>MOV R0, LOW(CNTTRA)</td>
</tr>
<tr>
<td>04A5</td>
<td>AH</td>
<td>=3874</td>
<td>MOV R0, LOW(CNTTRA)</td>
</tr>
<tr>
<td>=3875</td>
<td></td>
<td>=3876</td>
<td>MOV R0, LOW(CNTTRA)</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ  SRC STATEMENT

=3006 ;BKERR: BREAKPOINT LATCH WAS SET THROUGH BREAKPOINTS NOT ENABLED.
=3007 ;  DISPLAY HARDWARE ERROR MESSAGE
=3008 Dock: LDATA, $0CH
=3009 ;  JMP  ERROR
=300A ;
=300B CNTTRA: MMOV A, DSPTIM
=300C D920 =3100+  MMOV RL, DSPTIM
=300D F1 =3101+  MMOV A, #01
=300E 94F2 =3105 CALL  DELAY
=300F F4AF =3106 CALL  KBDPOL
=3010 F241 =3107  JB7  EPENT ;BY SET INDICATES NO KEYSTROKE.
=3108 ;
=3109 ;EPRUNS INPUT(KEY).
=3110 ;  IF KEY=END GO TO PARSER.
=3111 ;  INPUT KEY.
=3112 ;  IF KEY=NEXT GO TO PARSER.
=3113 ;  CONTINUE IN SAME MODE.
=3114 ;
=3015 14EC =3115 ;EPRUNS: CALL  INPKEY
=3016 FB =3116  MMOV A, KEY
=3017 D313 =3117  XR1 A, #KEYEND
=3018 96C7 =3118  JNZ  EPENT
=3019 14EC =3119 ;EPRUNS: CALL  INPKEY
=301A FB =3120  MMOV A, KEY
=301B D312 =3121  XR1 A, #KEYEND
=301C 96C7 =3122  JNZ  EPENT
=301D 2302 =3123  MMOV A, #2
=301E 3400 =3124  CALL  OUTUTL
=301F 8441 =3125  JMP  EPENT
=3126 ;
=3127 ;EPENT  EXECUTION MODE IS TO BE TERMINATED.
=3128 ;  JUMP INTO PARSER TO INTERPRET KEY ALREADY DETECTED.
=3027 0433 =3129 EPENT: JMP  MAIN2
=3130 ;
=3131 SIZECHK
=3029 =3134+ SIZE SET 201
=3135+;
=3136+;*******************************************************
=3145 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
## LOC  OBJ  LINE  SOURCE STATEMENT

<table>
<thead>
<tr>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0500</td>
<td>=3146</td>
<td>CODEBLK 115</td>
</tr>
<tr>
<td>0500</td>
<td>=3174</td>
<td>ORG  1200</td>
</tr>
<tr>
<td>0500</td>
<td>=3175</td>
<td>SAVE EP STATUS SAVE SUBROUTINE.</td>
</tr>
<tr>
<td>0500</td>
<td>=3176</td>
<td>FORCE CALL TO LOC 014H</td>
</tr>
<tr>
<td>0500</td>
<td>=3177</td>
<td>SAVE EP ACC:</td>
</tr>
<tr>
<td>0500</td>
<td>=3178</td>
<td>SAVE EP TIMER:</td>
</tr>
<tr>
<td>0500</td>
<td>=3179</td>
<td>SAVE EP PSW:</td>
</tr>
<tr>
<td>0500</td>
<td>=3180</td>
<td>SAVE EP R0:</td>
</tr>
<tr>
<td>0500</td>
<td>=3181</td>
<td>SAVE EP TOP-OF-STACK IN EP PC:</td>
</tr>
<tr>
<td>0500</td>
<td>=3182</td>
<td>RETURN</td>
</tr>
<tr>
<td>0500 744</td>
<td>=3183</td>
<td>SAVE EP CALL EPDK</td>
</tr>
<tr>
<td>0502 2383</td>
<td>=3184</td>
<td>MOV A, #3</td>
</tr>
<tr>
<td>0504 3400</td>
<td>=3185</td>
<td>CALL OUTUTL</td>
</tr>
<tr>
<td>050C 7451</td>
<td>=3186</td>
<td>CALL GVSWAP</td>
</tr>
<tr>
<td>0508 B081</td>
<td>=3187</td>
<td>MOV R0, #LOW(OVBS+OYSIZE)</td>
</tr>
<tr>
<td>0508 746A</td>
<td>=3188</td>
<td>CALL OVLDR</td>
</tr>
<tr>
<td>050C 8020</td>
<td>=3189</td>
<td>ORL P2, #000000008</td>
</tr>
<tr>
<td>050C 2314</td>
<td>=3190</td>
<td>MOV A, #14H</td>
</tr>
<tr>
<td>0510 91</td>
<td>=3191</td>
<td>MOVX @R1, A</td>
</tr>
<tr>
<td>0511 94DF</td>
<td>=3192</td>
<td>ANL P2, NOT #00000008</td>
</tr>
<tr>
<td>0513 B903</td>
<td>=3193</td>
<td>ORL P1, #000000011B</td>
</tr>
<tr>
<td>0515 F400</td>
<td>=3194</td>
<td>CALL EPSTEP</td>
</tr>
<tr>
<td>0517 8B20</td>
<td>=3195</td>
<td>ORL P2, #000100008</td>
</tr>
<tr>
<td>0519 94DF</td>
<td>=3196</td>
<td>ANL P2, NOT #0000100000</td>
</tr>
<tr>
<td>051B B803</td>
<td>=3197</td>
<td>ORL P1, #CINDEX OR ENADDR</td>
</tr>
<tr>
<td>051D F400</td>
<td>=3198</td>
<td>CALL EPSTEP</td>
</tr>
<tr>
<td>=3199</td>
<td></td>
<td>EXECUTION PROCESSOR IS NOW AT LOCATION 009H INTERNAL WITH</td>
</tr>
<tr>
<td>=3200</td>
<td></td>
<td>(RETURN ADDRESS+2) PUSHED ON STACK.</td>
</tr>
<tr>
<td>=3201</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=3202</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=3203</td>
<td>=3203</td>
<td>MOV R0, #LOW(OVBS+OYSIZE)</td>
</tr>
<tr>
<td>0521 7464</td>
<td>=3204</td>
<td>CALL OVLDR</td>
</tr>
<tr>
<td>0523 F400</td>
<td>=3205</td>
<td>CALL EPSSW</td>
</tr>
<tr>
<td>=3206</td>
<td>=3206</td>
<td>MOV EPACC, A</td>
</tr>
<tr>
<td>0525 B920</td>
<td>=3219</td>
<td>MOV R1, #EPACC</td>
</tr>
<tr>
<td>0527 A1</td>
<td>=3220</td>
<td>MOV R0, A</td>
</tr>
<tr>
<td>0528 F400</td>
<td>=3224</td>
<td>CALL EPSSW</td>
</tr>
<tr>
<td>=3225</td>
<td>=3225</td>
<td>MOV EPTIMR, A</td>
</tr>
<tr>
<td>052A B922</td>
<td>=3238</td>
<td>MOV R1, #EPTIMR</td>
</tr>
<tr>
<td>052C A1</td>
<td>=3239</td>
<td>MOV R0, A</td>
</tr>
<tr>
<td>052D F400</td>
<td>=3243</td>
<td>CALL EPSSW</td>
</tr>
<tr>
<td>=3244</td>
<td>=3244</td>
<td>MOV EPSSW, A</td>
</tr>
<tr>
<td>052F B521</td>
<td>=3257</td>
<td>MOV R1, #EPSSW</td>
</tr>
<tr>
<td>0531 A1</td>
<td>=3258</td>
<td>MOV R0, A</td>
</tr>
<tr>
<td>0532 F400</td>
<td>=3262</td>
<td>CALL EPSSW</td>
</tr>
<tr>
<td>=3263</td>
<td>=3263</td>
<td>MOV EPB, A</td>
</tr>
<tr>
<td>0534 B923</td>
<td>=3276</td>
<td>MOV R1, #EPB</td>
</tr>
<tr>
<td>0536 A1</td>
<td>=3277</td>
<td>MOV R0, A</td>
</tr>
<tr>
<td>0537 D888</td>
<td>=3281</td>
<td>MOV R0, #LOW(OVBS+OYSIZE)</td>
</tr>
<tr>
<td>0539 746A</td>
<td>=3282</td>
<td>CALL OVLDR</td>
</tr>
<tr>
<td>=3283</td>
<td>=3283</td>
<td>MOV R1, #EPSSW</td>
</tr>
<tr>
<td>053B B921</td>
<td>=3292</td>
<td>MOV R1, #EPPSW</td>
</tr>
<tr>
<td>053D F1</td>
<td>=3293</td>
<td>MOV R0, #1</td>
</tr>
<tr>
<td>053E 07</td>
<td>=3297</td>
<td>DEC A</td>
</tr>
<tr>
<td>053F 5387</td>
<td>=3298</td>
<td>ANL R0, #07H</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>8541</td>
<td>E7</td>
<td>=3299</td>
<td>RL</td>
<td>A</td>
</tr>
<tr>
<td>8542</td>
<td>8388</td>
<td>=3300</td>
<td>ADD</td>
<td>R0,#8H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3301</td>
<td>MMMOV</td>
<td>#A1,#DLO</td>
</tr>
<tr>
<td>8544</td>
<td>E939</td>
<td>=3314+</td>
<td>MOV</td>
<td>RL,#DLO</td>
</tr>
<tr>
<td>8546</td>
<td>E1</td>
<td>=3315+</td>
<td>MOV</td>
<td>#R0,#1</td>
</tr>
<tr>
<td>8547</td>
<td>F4B7</td>
<td>=3319</td>
<td>CALL</td>
<td>EFFET</td>
</tr>
<tr>
<td>8549</td>
<td>03FE</td>
<td>=3320</td>
<td>ADD</td>
<td>R0,#-2</td>
</tr>
<tr>
<td>854B</td>
<td>AA</td>
<td>=3321</td>
<td>MOV</td>
<td>LDATA,A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3322</td>
<td>MMMOV</td>
<td>#PPCL0,A</td>
</tr>
<tr>
<td>854C</td>
<td>BS24</td>
<td>=3325+</td>
<td>MOV</td>
<td>RL,#PPCL0</td>
</tr>
<tr>
<td>854E</td>
<td>R1</td>
<td>=3336+</td>
<td>MOV</td>
<td>#RLA</td>
</tr>
<tr>
<td>854F</td>
<td>F4C3</td>
<td>=3346</td>
<td>CALL</td>
<td>EPSTOR</td>
</tr>
<tr>
<td>8551</td>
<td>BS38</td>
<td>=3347</td>
<td>MOV</td>
<td>RL,#DLO</td>
</tr>
<tr>
<td>8553</td>
<td>11</td>
<td>=3348</td>
<td>INC</td>
<td>#R1</td>
</tr>
<tr>
<td>8554</td>
<td>F4B7</td>
<td>=3349</td>
<td>CALL</td>
<td>EFFET</td>
</tr>
<tr>
<td>8556</td>
<td>AR</td>
<td>=3350</td>
<td>MOV</td>
<td>LDATA,A</td>
</tr>
<tr>
<td>8557</td>
<td>53F0</td>
<td>=3351</td>
<td>ANL</td>
<td>R0,#11110B</td>
</tr>
<tr>
<td>8559</td>
<td>0A</td>
<td>=3352</td>
<td>XCH</td>
<td>R0,LDATA</td>
</tr>
<tr>
<td>855A</td>
<td>13FF</td>
<td>=3353</td>
<td>ADDC</td>
<td>R0,#-1</td>
</tr>
<tr>
<td>855C</td>
<td>53F8</td>
<td>=3354</td>
<td>ANL</td>
<td>R0,#000111B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3355</td>
<td>MMMOV</td>
<td>#PPCHI,A</td>
</tr>
<tr>
<td>855E</td>
<td>BS25</td>
<td>=3356+</td>
<td>MOV</td>
<td>RL,#PPCHI</td>
</tr>
<tr>
<td>8560</td>
<td>R1</td>
<td>=3360+</td>
<td>MOV</td>
<td>#RLA</td>
</tr>
<tr>
<td>8561</td>
<td>0A</td>
<td>=3361</td>
<td>ORL</td>
<td>R0,LDATA</td>
</tr>
<tr>
<td>8562</td>
<td>AR</td>
<td>=3362</td>
<td>MOV</td>
<td>LDATA,A</td>
</tr>
<tr>
<td>8563</td>
<td>F4C3</td>
<td>=3363</td>
<td>CALL</td>
<td>EPSTOR</td>
</tr>
<tr>
<td>8565</td>
<td>BS25</td>
<td>=3364</td>
<td>MOV</td>
<td>R0,#PPCHI</td>
</tr>
<tr>
<td>8567</td>
<td>347C</td>
<td>=3365</td>
<td>CALL</td>
<td>UDLOAD1</td>
</tr>
<tr>
<td>8569</td>
<td>2340</td>
<td>=3366</td>
<td>MOV</td>
<td>R0,#01000000B ;** FOR DISPLAY</td>
</tr>
<tr>
<td>856B</td>
<td>0408</td>
<td>=3367</td>
<td>CALL</td>
<td>WDISP</td>
</tr>
<tr>
<td>856D</td>
<td>BS20</td>
<td>=3368</td>
<td>MOV</td>
<td>R0,#EPACC</td>
</tr>
<tr>
<td>856F</td>
<td>3400</td>
<td>=3369</td>
<td>CALL</td>
<td>DPMID</td>
</tr>
<tr>
<td>8571</td>
<td>03</td>
<td>=3370</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3371</td>
<td>SIZECHK</td>
<td></td>
</tr>
<tr>
<td>0872</td>
<td></td>
<td>=3380+</td>
<td>SIZE</td>
<td>SLT 114</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3381+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3382+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3383+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3384+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3385+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3386+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=3387+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC 0EJ  LINE  SOURCE STATEMENT

3348 $  INCLUDE(+-0:HEXFILE.MOD)
0000  =3333  CHARCP  EQU  00H  ;<CR>
0001  =3334  CHRSLF  EQU  09H  ;<LF>
001A  =3395  CTRLZ  EQU  1AH  ;CONTROL-Z
      =3396 ;
      =3397  CODEBLK  00
0297  =3412+  ORG  063
      =3416  ;IRECIN HEXFILE RECORD INPUT ROUTINE
0297  34CD  =3417  IRECIN  CALL  CHARIN
0299  D31A  =3418  XRL  A,#CTRLZ
029B  C6E9  =3419  JZ  DONE
029D  D31A  =3420  XRL  A,#CTRLZ
029F  D32A  =3421  XRL  A,`(`?)`
0301  9697  =3422  JNZ  IRECIN
      =3423  MOV  CHKSUM,ZERO
0303  0D08  =3424+  MOV  CHKSUM,#ZERO
0305  14F0  =3432  CALL  BYTEIN
      =3433  MOV  BUFNT,A
0307  D941  =3446+  MOV  RL,#BUFNT
0309  A1  =3447+  MOV  ARL,A
030B  14F0  =3451  CALL  BYTEIN
      =3452  MOV  SMHAL,A
030D  D931  =3455+  MOV  RL,#SMHAL
030F  A1  =3466+  MOV  ARL,A
0311  14F0  =3470  CALL  BYTEIN
      =3471  MOV  SMARL,A
0313  D939  =3484+  MOV  RL,#SMARL
0315  A1  =3495+  MOV  ARL,A
0317  14F0  =3499  CALL  BYTEIN
      =349A  MOV  RECVP,A
0319  D942  =3503+  MOV  RL,#RECVP
031B  A1  =3504+  MOV  ARL,A
      =3508 ;
      =3509 ;HDATIN HEX DATA BYTE IN
      =3510  HDATIN:  MOV  A,BUFNT
031D  D941  =3519+  MOV  RL,#BUFNT
031F  F1  =3520+  MOV  A,ARL
0321  C6CC  =3524  JZ  RECOND
0323  14F0  =3525  CALL  BYTEIN
0325  9A  =3526  MOV  LDATA,A
0327  F400  =3527  CALL  LSTORE
0329  34F2  =3528  CALL  INCRA
      =3529  MDEC  BUFNT
032B  D941  =3534+  MOV  RL,#BUFNT
032D  F1  =3535+  MOV  A,#RL
032F  87  =3539  DEC  A
0331  A1  =3544+  MOV  ARL,A
0333  4409  =3547  JMP  HDATIN
      =3548 ;
0335  C6CD  =3549  RECOND:  CALL  CHARIN
0337  D33F  =3550  XRL  A,`(`?)`
0339  6608  =3551  JZ  CKSMOK
033B  D33F  =3552  XRL  A,`(`?)`
      ;SWITCH BACK TO DATA CHARACTER
033D  3488  =3553  CALL  NIBIN2
      ;JOIN SUBROUTINE ALREADY IN PROGRESS
033F  14F2  =3554  CALL  BYTEIN
      ;DITO
LOC OBJ LINE SOURCE STATEMENT

=3555 (RESULT FOR NON-"?" CHARACTERS IS AS IF
=3556 BYTE IN WAS CALLED.)
=3557 MMOV A, CHKSUM
0208 FD =3573+ MOV A, CHKSUM
=3577 JNZ CHKERR
0209 96E1 =3578 CKSUM: MMOV A, KECVT
020E 8942 =3587+ MOV SL, KECVT
020D F1 =3588+ MOV N, ORA
020E 007 =3592 JZ HRECIN
=3593;
=3594: DONE HEX FILE CORRECTLY RECEIVED
020E 83 =3595 DONE: RET
=3596;
=3597: CHKERR CHECKSUM ERROR IN INPUT RECORD DETECTED
020E 8A0C =3598 CHKERR: MOV LDATA, #6CH
020E 294A =3599 JMP FLROR
=3600 SIZECHK
004E =3803+ SIZE SET 78
=3804;
=3805+*******************************************************
=3614;
=3615 CODEBLK 12
06F0 =3620+ ORG 248
=3624: BYTE INPUT SUBROUTINE.
=3625; RECEIVES TWO HEXDECIMAL CHARACTERS FROM THE TAPE INPUT DEVICE
=3626; AND ASSEMBLES THEM INTO A SINGLE BYTE OF DATA.
0060 34B8 =3627 BYTEIN: CALL NIBIN
0062 47 =3628 BYTEIN1: SWAP A
0063 9A =3629 MOV LDATA, A
0064 880B =3630 CALL NIBIN
=3631 MRL LDATA, A
0066 4A =3640+ ORL A, LDATA
0067 9A =3650+ MOV LDATA, A
0069 06 =3660 ADD A, CHKSUM
006A 0D =3665 MOV CHKSUM, A
006E 8A =3666 MOV A, LDATA
006F 83 =3667 RET
=3668 SIZECHK
008C =3671+ SIZE SET 12
=3672;
=3673+*******************************************************
=3682;
=3683 CODEBLK 25
0188 =3693+ ORG 440
=3697: NIBIN RECOEIVES A HEXDECIMAL CHARACTERS AND PRODUCES A MASKED FOUR-BIT VALUE.
=3698; NOTE: ERROR CHECKING DONE TO VERIFY HEXDECIMAL VALIDITY
0188 34CD =3699 NIBIN: CALL CHARIN
0188 83C6 =3700 NIBIN2: ADD A, #3AH
=3701 ;ACC=8F6-8FF FOR CHARACTERS '9'-"9"
0188 83C9 =3702 JNC NIBIN3
=3703 ;CHARACTERS > '9' PRODUCE OVERFLOW
018C 66C2 =3704 JNC NIBIN3
=3705 ;ACC=8F6-8FH FOR CHARACTERS '9'-"9"
018E 83F9 =3706 JNC ASCLAR
=3707 ;ERROR IF CHARACTER BETWEEN '9' AND 'A'
=3708;
=3709;
<table>
<thead>
<tr>
<th>LOC</th>
<th>ODF</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>01C2</td>
<td>03F8</td>
<td>=3788</td>
<td>NIB3: ADD A, 0-6 ; ACC=0FH+0FH FOR CHARACTERS '0'-'F'</td>
</tr>
<tr>
<td>01C4</td>
<td>0310</td>
<td>=3790</td>
<td>ADD A, 018H ; ACC=00H-0FH FOR CHARACTERS '0'-'F';</td>
</tr>
<tr>
<td>01C6</td>
<td>E6C9</td>
<td>=3711</td>
<td>JNC ASCERR</td>
</tr>
<tr>
<td>01C8</td>
<td>02</td>
<td>=3712</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td>3713</td>
<td>=3714</td>
<td>ASCERR ILLEGAL HEXADECIMAL CHARACTER RECEIVED</td>
</tr>
<tr>
<td>01C9</td>
<td>0188</td>
<td>=3715</td>
<td>ASCERR: MOV DAA, #0AH</td>
</tr>
<tr>
<td>01CD</td>
<td>249A</td>
<td>=3716</td>
<td>JMP PEERRR</td>
</tr>
<tr>
<td>04d5</td>
<td>SKF</td>
<td>=3720m SIZE SET 21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=3721+</td>
<td>=3722+</td>
<td>************************************************************</td>
</tr>
<tr>
<td></td>
<td>=3723</td>
<td>=3724</td>
<td>CODEBLK 5</td>
</tr>
<tr>
<td>01CD</td>
<td>=3743+</td>
<td>ORG 461</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=3747</td>
<td>; CHAIRN CHARACTER INPUT ROUTINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=3748</td>
<td>; RECEIVES ONE ASCII CHARACTER FROM THE LOGICAL READER DEVICE</td>
<td></td>
</tr>
<tr>
<td>01D0</td>
<td>0449</td>
<td>=3749</td>
<td>CALL CIN</td>
</tr>
<tr>
<td>01DF</td>
<td>537F</td>
<td>=3750</td>
<td>ANL A, #7FH</td>
</tr>
<tr>
<td>01D1</td>
<td>83</td>
<td>=3751</td>
<td>RET</td>
</tr>
<tr>
<td>0005</td>
<td>=3752</td>
<td>SIZECHK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=3755+</td>
<td>SIZE SET 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=3756+</td>
<td>=3757+</td>
<td>************************************************************</td>
</tr>
<tr>
<td></td>
<td>=3758+</td>
<td>=3759+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=3760</td>
<td>=3761</td>
<td></td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC 067          LINE          SOURCE STATEMENT

0572             =3760          CODELX 100

0572             =3794+         ORG 1394

0572             =3798 ;HFILD HEX FILE OUTPUT SUBROUTINE

0572             =3799 ; WHEN CALLED WITH F=0 OUTPUT IS STANDARD HEX FILE FORMAT

0572             =3800 ; WHEN CALLED WITH F=1 OUTPUT IS FORMATTED DATA DUMP TO CRT

0572             =3801 HFILD. MMOV MEMHL, SMARH

0572  B931       =3817+         MOV RL, SMARH

0574  F1        =3818+         MOV A, RY1

0575  B935       =3824+         MOV RL, MERNII

0577  R1        =3825+         MOV ARL, A

0577  B930       =3820          MMOV MEMLO, SMALD

0577  B934       =3844+         MOV RL, SMALD

0579  F1        =3845+         MOV A, RY1

0579  B934       =3851+         MOV RL, MERNLO

0579  R1        =3852+         MOV ARL, A

057E  0000       =3855          MMOV CHRSUM, ZERO

0580  B865       =3864          MOV R8, HEXBUF

0585 ;            =3865 ; LDBYTE LOAD NEXT BYTE FROM MEMORY INTO HEX BUFFER

0582  14FC       =3867         LDBYTE: CALL LFETCH

0584  FR        =3868          MOV A, LDATA

0585  R0        =3869          MOV A, R0

0585  18        =3870          INC R0

0586  B4E2       =3871          CALL CMNPS

0586  E606       =3872          JAC ENDIFIL

0588  34F2       =3873          CALL INCMA

058D  F0        =3874          MOV A, R0

058E  03B0       =3875          ADD A, A-

0590  E682       =3876          JNC LDBYTE

0592  D400       =3877          CALL HRECO

0594  A472       =3878          JMP HFILD

0595 ;            =3879 ;

0596  D400       =3880 ; ENDIFIL END HEX FILE TRANSMISSION;

0598  B697       =3881+         PRINT OUT BUFFER FOR LAST DATA RECORD

0599  34D2       =3882 ; PRINT OUT CANNED 'END-OF-FIIE' RECORD

059A  18         =3883 ; RETURN.

059C  B8AE       =3884 ENDIFIL CALL HRECO

059F  B875       =3885          JF0 HFDONE

059F  34D2       =3886          CALL TCRILFO

059C B8AE       =3887          MOV R0, #(LOW EOFREC)

059E  F8        =3888 ENDIF1: MOV A, R0

059F  R3        =3889          MOV A, R0

059F  CG07       =3890          JZ HFDONE

05AE  D48D       =3891          CALL CHAEO

05A4  18        =3892          INC R0

05A5  A49E       =3893          JMP ENDF1

05A7  34D2       =3894 HFDONE: CALL TCRILFO

05A9  231A       =3895          MOV A, #NTRLZ

05AB  D48D       =3896          CALL CHAEO

05AD  03        =3997          RET

05AE  203A3030   =3998 ; EOFREC CHARACTER SKETING FOR CANNED END-OF-FIIE RECORD FOR

05AE  203A3030   =3999 ; INTEL HEX FILE FORMAT STANDARD.

05AE  203A3030   =3998 ; EOFREC: DD '00000001T'

All mnemones copyrighted © Intel Corporation 1976.
LOC OBJ    LINE SOURCE STATEMENT

0582 30303030
0588 30314646

059A 00 =3902 DB 0 ; END OF STRING CODE BYTE
         =3903 SIZECHK  
0849 =3904: SIZE SET 73
         =3907+:
         =3908+************************************************************
         =3917 :   
         =3918 ;
         =3919 CODEBLK 90
         =391A+ ORG 1536
         =391B : HEXRECO HEXDECIMAL RECORD OUTPUT SEQUENCE.
         =391C : HEX BUFFER ALREADY LOADED.
0600 F8 =391D HEXRECO MOV A, R0
0601 39B =391E ADD A, #HEXBUF
0602 B941 =391F MOV R1, #BUFCNT
0605 A1 =3920 MOV @R1, A
0606 3402 CALL TCRLO
0608 2320 MOV A, #'
060A 048D =3922 CALL CHAD
060C 0617 =3923 JF #FDUMP1
060E 2330 =3924 MOV A, #'
0610 048D =3925 CALL CHAD
0612 B941 =3926 MOV R1, #BUFCNT
0614 F1 =3927 MOV A, #AK1
0615 340C =3928 CALL BYTEO
         =3929 FDUMP1: MOV A, #MEM1
0617 3935 =4005: MOV R1, #MEM1
0619 F1 =4006: MOV A, #AK1
061A 340B =4007 CALL BYTEO
         =4008 MMOV A, #MEMLO
061C 3934 =4020: MOV R1, #MEMLO
061E F1 =4021: MOV A, #AK1
061F 340B =4025 CALL BYTEO
0621 BC28 =4026 JF #FDUMP2
0623 27 =4027 CLR A
0624 340B =4028 CALL BYTEO
0626 C42C =4029 JMP DATO
0628 233D =4030 FDUMP2: MOV A, #"=
062A B48D =4031 CALL CHAD
         =4032 ;DATO DATA OUTPUT
062C B865 =4033 DATO: MOV R0, #HEXBUF
062C BE32 =4034 DATO1: JF #FDUMP5
0630 C436 =4035 JMP FDUMP3
0632 2320 =4036 FDUMP5: MOV A, #'
0634 B48D =4037 CALL CHAD
0636 F0 =4038 FDUMP3: MOV A, #R88
0637 340B =4039 CALL BYTEO
0639 18 =4040 INR R0
         =4041 MO.RNZ BUFCT, DATO1
063A B941 =4046: MOV R1, #BUFCNT
063C F1 =4047: MOV A, #AK1
063D 07 =4051: DEC A

All mnemonics copyrighted © Intel Corporation 1976.
LOC  OBJ  LINE  SOURCE STATEMENT

063E  A1   =4056+  MOV  @RL.A
063F  062E  =4058+  JNZ  DAT01

0663  ;  ENDREC ENDCODE TRANSMITTED
0641  B648  =4064+  ENDCODE:  JF0  FDUMP4
0645  9605  =4065+  MMOV  A, CHRSUM
0646  FD   =4061+  MOV  A, CHRSUM
0644  37   =4065+  CPL  A
0645  17   =4066+  INC  A
0647  3408  =4067+  CALL  BYTEO
0648  83   =4068+  FDUMP4:  RET

0049  =4089+  SIZECHK

0100  =4104+  CODEBLK  9
0102  =4114+  ORG  466

0102  230D  =4119+  TCRLFO:  MOV  A, CHARCR
0104  B4BD  =4128+  CALL  CHARRD
0106  230H  =4121+  MOV  A, CHARLF
0108  B4BD  =4122+  CALL  CHARRD
010A  83   =4123+  RET

0089  =4129+  SIZE  SET  9

0100  =4138+  CODEBLK  11
0102  =4149+  ORG  475

010B  820D  =4155+  BYTEO:  MOV  LDATA.A
010C  6D   =4155+  ADD  A, CHRSUM
010D  BD   =4156+  MOV  CHRSUM.A
010E  FA   =4157+  MOV  A, LDATA
010F  47   =4158+  SWAP  A
0110  84D8  =4159+  CALL  NIBO
0112  FA   =4160+  MOV  A, LDATA
0113  84D8  =4161+  CALL  NIBO
0115  83   =4162+  RET

00BB  =4166+  SIZE  SET  11

0116  =4177+  CODEBLK  12
0116  =4187+  ORG  486

0116  530F  =4193+  HEXASC:  ANL  A, 00H
0118  83F6  =4194+  ADD  A, #(-10)
011A  F6EF  =4195+  JC  HDXNIB
011C  832A  =4196+  ADD  A, #(10H-'0')
011E  83   =4197+  RET
011F  8341  =4198+  HDXNIB:  ADD  A, #('A')

All mnemonics copyrighted © Intel Corporation 1976.

1-132
```
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>01F1 83</td>
<td>=4199</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>=4200</td>
<td>SIZECHK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000C</td>
<td>=4203+</td>
<td>SIZE SET 12</td>
<td></td>
</tr>
<tr>
<td>=4204+;</td>
<td>=4205+;</td>
<td>************************************************************</td>
<td></td>
</tr>
<tr>
<td>=4214 ;</td>
<td>=4215 ;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4216 DECLARE BITSO,CONST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000B</td>
<td>=4220 BITSO EQU 11 ; DATA BITS PUT OUT (INCLUDING TWO STOP BITS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4231 ;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4232</td>
<td>CODEBLK 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04C9</td>
<td>=4252+</td>
<td>ORG 1225</td>
<td></td>
</tr>
<tr>
<td>=4256 ;</td>
<td>HBDLAY HALF-BIT TIME DELAY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4257 HBDLAY MMOV R1.HBITI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04C9 B27</td>
<td>=4273+</td>
<td>MOV R1.HBITI</td>
<td></td>
</tr>
<tr>
<td>04CB F1</td>
<td>=4274+</td>
<td>MOV R1.R1</td>
<td></td>
</tr>
<tr>
<td>04CC B945</td>
<td>=4280+</td>
<td>MOV R1.RH</td>
<td></td>
</tr>
<tr>
<td>04CE A1</td>
<td>=4281+</td>
<td>MOV R1.A</td>
<td></td>
</tr>
<tr>
<td>=4284</td>
<td>MMOV R1.HBITLO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04CF B926</td>
<td>=4300+</td>
<td>MOV R1.HBITLO</td>
<td></td>
</tr>
<tr>
<td>04D0 F1</td>
<td>=4301+</td>
<td>MOV R1.R1</td>
<td></td>
</tr>
<tr>
<td>04D2 A9</td>
<td>=4314+</td>
<td>MOV R1.A</td>
<td></td>
</tr>
<tr>
<td>04D3 8407</td>
<td>=4317</td>
<td>JMP HBD1</td>
<td></td>
</tr>
<tr>
<td>04D5 B900</td>
<td>=4318 HBD2 ; MOV R1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04D7 E907</td>
<td>=4319 HBD1 ; DJNZ R1.HBD1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4320</td>
<td>MDJNZ H.HBD2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04D9 B945</td>
<td>=4325+</td>
<td>MOV R1.RH</td>
<td></td>
</tr>
<tr>
<td>04DB F1</td>
<td>=4326+</td>
<td>MOV R1.R1</td>
<td></td>
</tr>
<tr>
<td>04DC 87</td>
<td>=4338+</td>
<td>DEC A</td>
<td></td>
</tr>
<tr>
<td>04DD A1</td>
<td>=4335+</td>
<td>MOV R1.A</td>
<td></td>
</tr>
<tr>
<td>04DE 9625</td>
<td>=4339+</td>
<td>JNZ HBD2</td>
<td></td>
</tr>
<tr>
<td>04E0 83</td>
<td>=4341</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>=4342</td>
<td>SIZECHK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018</td>
<td>=4345+</td>
<td>SIZE SET 24</td>
<td></td>
</tr>
<tr>
<td>=4346+;</td>
<td>=4347+;</td>
<td>************************************************************</td>
<td></td>
</tr>
<tr>
<td>=4356 ;</td>
<td>=4357 #EXEC1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
LOC OBJ LINE SOURCE STATEMENT

=4358 CODEBLK 48
=4383+ ORG 1467
=4387 ; NIDO MASK ACC TO MAKE HEX NIBBLE, TRANSLATE TO ASCII AND OUTPUT
=4388 NIDO: CALL HEXASC
=4389 ;
=4390 ; CHARD: CONSOLE OUTPUT SUBROUTINE
=4391 ; WRITE THE CONTENTS OF THE ACC TO THE CRT DISPLAY SCREEN
=4392 CHARD: MMOV REGC,A

=4485+ MOV RL, #REGC
=4486+ MOV ORL .A
=4418 MMOV B,BITSO ; SET NUMBER OF BITS TO BE TRANSMITTED
=4421+ MOV RL, #B
=4422+ MOV ORL, #BITSO
=4427 C0L: JC CO2
=4428 ANL PL, #NOT TTYOUT
=4429 JMP C03
=4430 CO2: ORL PL, #TTYOUT
=4431 NOP ; EVEN OUT TWO BRANCH EXECUTION TIMES
=4432 NOP
=4433 C03: CALL HDBLAY
=4434 CALL HDBLAY
=4435 CLR C ; SET WHAT WILL EVENTUALLY BECOME A STOP BIT
=4436 CFI C
=4437 MMRC REGC ; ROTATE CHARACTER RIGHT ONE BIT,
=4442+ MOV RL, #REGC
=4443+ MOV A, #RL
=4444+ RRC A
=4452+ MOV ORL, #A
=4455 MOVE NEXT DATA BIT INTO CARRY
=4461+ MOJNZ B, CO1 ; CHECK IF CHARACTER (AND STOP BIT(S)) DONE
=4462+ MOV A, #RL
=4463+ MOV A, #RL
=4466+ DEC A
=4471+ MOV ORL, #A
=4475+ JNZ CO1
=4477 RET
=4478 SIZECHK
=4481+ SIZE SET 39
=4482+;
=4483+;*******************************************************************************
=4492 ;
=4493 CODEBLK 47
=4523+ ORG 1689
=4527 ; Cin: CONSOLE INPUT SUBROUTINE WAITS FOR A KEYSTROKE AND
=4528 ; RETURNS WITH 8 BITS IN REG ACC.
=4529 CIN: MOV RL, #B
=4530 MOV ORL, #C ; DATA BITS TO BE READ
=4531 C10: JNT1 C10
=4532 JNT1 C10
=4533 C11: JNT1 C11
=4534 JNT1 C11
=4535 CALL HDBLAY
=4536 JT1 C11
=4537 C12: CALL HDBLAY

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0658</td>
<td>94C9</td>
<td>=4538</td>
<td>CALL HBOLAY</td>
</tr>
<tr>
<td>065D</td>
<td>5662</td>
<td>=4539</td>
<td>JT1 CI3 ;CHECK SID LINE LEVEL</td>
</tr>
<tr>
<td>065F</td>
<td>97</td>
<td>=4540</td>
<td>CLR C ;DATA BIT IN CY</td>
</tr>
<tr>
<td>0668</td>
<td>C465</td>
<td>=4541</td>
<td>JMP CI4</td>
</tr>
<tr>
<td>0662</td>
<td>97</td>
<td>=4542</td>
<td>CLR C</td>
</tr>
<tr>
<td>0663</td>
<td>R7</td>
<td>=4543</td>
<td>CPL C</td>
</tr>
<tr>
<td>0664</td>
<td>00</td>
<td>=4544</td>
<td>NOP ;EVEN OUT BRANCH EXECUTION TIMES</td>
</tr>
<tr>
<td>0665</td>
<td>00</td>
<td>=4545</td>
<td>CI4: NOP</td>
</tr>
<tr>
<td>0666</td>
<td>00</td>
<td>=4546</td>
<td>NOP</td>
</tr>
<tr>
<td>0667</td>
<td>00</td>
<td>=4547</td>
<td>NOP</td>
</tr>
<tr>
<td>0668</td>
<td>B944</td>
<td>=4548</td>
<td>MRRC REGC</td>
</tr>
<tr>
<td>066A</td>
<td>F1</td>
<td>=4553+</td>
<td>MOV R1,#REGC</td>
</tr>
<tr>
<td>066B</td>
<td>67</td>
<td>=4554+</td>
<td>MOV A,#R1</td>
</tr>
<tr>
<td>066C</td>
<td>R1</td>
<td>=4558+</td>
<td>RRC A</td>
</tr>
<tr>
<td>066D</td>
<td>B943</td>
<td>=4563+</td>
<td>MOV A,#R1</td>
</tr>
<tr>
<td>066F</td>
<td>F1</td>
<td>=4564</td>
<td>MOV A,#R1</td>
</tr>
<tr>
<td>0670</td>
<td>07</td>
<td>=4565+</td>
<td>DEC A</td>
</tr>
<tr>
<td>0671</td>
<td>R1</td>
<td>=4566+</td>
<td>MOV A,#R1</td>
</tr>
<tr>
<td>0672</td>
<td>9659</td>
<td>=4567+</td>
<td>JNZ CI2</td>
</tr>
<tr>
<td>0674</td>
<td>B944</td>
<td>=4568+</td>
<td>MOV R1,#REGC</td>
</tr>
<tr>
<td>0676</td>
<td>F1</td>
<td>=4569+</td>
<td>MOV A,#R1</td>
</tr>
<tr>
<td>0677</td>
<td>83</td>
<td>=456A</td>
<td>RET ;CHARACTER COMPLETE</td>
</tr>
<tr>
<td>067F</td>
<td>=4682</td>
<td>SIZE CHK</td>
<td></td>
</tr>
<tr>
<td>=4685+</td>
<td>SIZE SET 47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4686+;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4687+;</td>
<td>*********************************</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=4616</td>
<td>$EJECT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ
LINE SOURCE STATEMENT

4617 $ INCLUDE (:OR-MEIBRT.MOD).
4618 CODEBLK 15
02E5
4633+ ORG 741
4637 ;COMIL COMMAND TO fill ADDRESS SPACE BETWEEN SRA AND EMA WITH DATA
4638 ; IN LOW BYTE OF MEN,
4639 COMIL: MMOT LDATA.MEMIO
02E9 B934
4635+ MOV RL #MEMLO
02E7 F1
4656+ MOV A, #R1
02E9 44E9
4669+ MOV LDATA.A
02E9 FF30
4672 LFILL: CALL LSTORE
02E9 D4E2
4673 CALL CMPMOS
02ED 86F3
4674 JNC LFILL1
02EF 34F2
4675 CALL INCSPM
02FI 44E9
4676 JMP LFILL
02F3 83
4677 LFILL1: RET
4678 SIZECHK
00E8
4681+ SIZE SET 15
4682+;
4683+ ;******************************************************************************
4692 ;
4693 CODEBLK 4
00FC
4698+ ORG 252
4702 ; FETCH FETCHS CONTENTS OF LOGICAL MEMORY ADDRESS DETERMINED BY
4703 ; (TYPE), (SMAHI), & (SMALO) INTO LDATA.
00FC D47B
4704 LFETCH: CALL AFETCH
00FE AA
4705 MOV LDATA.A
00FF 83
4706 RET
4707 SIZECHK
0004
4710+ SIZE SET 4
4711+;
4712+ ;******************************************************************************
4721 ;
4722 CODEBLK 75
0678
4724+ ORG 156
4756 ;
4757 ; AFETCH LOGICAL FETCH SUBROUTINE
4758 ; FETCHS CONTENTS OF VARIOUS MEMORY SPACES TO ACC.
4759 AFETCH: MMOV A, TYPE
0678 B937
4768+ MOV RL #TYPE
0679 F1
4769+ MOV A, #R1
0679 B37E
4773 ADD A, #LOW LFETBL
0679 B3
4774 JMPF A
4775 ;
067E 04
4776 LFETBL: DB LOW LFEMP
067F 98
4777 DB LOW LFEDM
0680 00
4778 DB LOW LFEBREG
0681 8C
4779 DB LOW LFTEINT
0683 00
4780 DB LOW LFEBRK
0683 01
4781 DB LOW LFEDBRK
4782 ;
4783 LFEMP: MMOV A, SMAHI
0684 0931
4792+ MOV RL #SMHI
0686 F1
4793+ MOV A, #R1
0687 9690
4797 JNZ LFEDM
4798 MMOV A, SMALO
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0689</td>
<td>8930</td>
<td>4887</td>
<td>MOV RL. #SMAL0</td>
</tr>
<tr>
<td>0689</td>
<td>F1</td>
<td>4888</td>
<td>MOV A. #R1</td>
</tr>
<tr>
<td>068C</td>
<td>83E9</td>
<td>4812</td>
<td>ADD A. #O-OSIZE</td>
</tr>
<tr>
<td>068E</td>
<td>F098</td>
<td>4813</td>
<td>JC LFEDM</td>
</tr>
<tr>
<td>0690</td>
<td>B930</td>
<td>4814</td>
<td>MMov A. #SMALO</td>
</tr>
<tr>
<td>0692</td>
<td>F1</td>
<td>4823</td>
<td>MOV RL. #SMALO</td>
</tr>
<tr>
<td>0693</td>
<td>83E9</td>
<td>4824</td>
<td>MOV A. #R1</td>
</tr>
<tr>
<td>0695</td>
<td>99</td>
<td>4826</td>
<td>ADD A. #OBUF</td>
</tr>
<tr>
<td>0696</td>
<td>F1</td>
<td>4829</td>
<td>MOV RL. A</td>
</tr>
<tr>
<td>0697</td>
<td>03</td>
<td>4834</td>
<td>RET</td>
</tr>
<tr>
<td>0698</td>
<td>94E1</td>
<td>4832</td>
<td>CALL LPSHEL</td>
</tr>
<tr>
<td>0699</td>
<td>81</td>
<td>4833</td>
<td>MOVX A. #R1</td>
</tr>
<tr>
<td>0699</td>
<td>83</td>
<td>4834</td>
<td>KLT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4835</td>
<td>;</td>
</tr>
<tr>
<td>069C</td>
<td>B930</td>
<td>4845</td>
<td>MOV RL. #SMAL0</td>
</tr>
<tr>
<td>069E</td>
<td>F1</td>
<td>4846</td>
<td>MOV A. #R1</td>
</tr>
<tr>
<td>069F</td>
<td>537F</td>
<td>4850</td>
<td>ANL A. #01111111B</td>
</tr>
<tr>
<td>0698</td>
<td>G695</td>
<td>4851</td>
<td>J2 LFED0</td>
</tr>
<tr>
<td>0693</td>
<td>8487</td>
<td>4852</td>
<td>JMP LPEET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4853</td>
<td>;</td>
</tr>
<tr>
<td>06A5</td>
<td>B923</td>
<td>4854</td>
<td>MMov A. #PR0</td>
</tr>
<tr>
<td>06A7</td>
<td>F1</td>
<td>4856</td>
<td>MOV RL. #PR0</td>
</tr>
<tr>
<td>06A8</td>
<td>03</td>
<td>4858</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4859</td>
<td>;</td>
</tr>
<tr>
<td>06A9</td>
<td>B930</td>
<td>4869</td>
<td>MOV RL. #SMAL0</td>
</tr>
<tr>
<td>06AB</td>
<td>F1</td>
<td>4880</td>
<td>MOV A. #R1</td>
</tr>
<tr>
<td>06AC</td>
<td>8320</td>
<td>4884</td>
<td>ADD A. #EPACC</td>
</tr>
<tr>
<td>06AE</td>
<td>99</td>
<td>4885</td>
<td>MOV RL. A</td>
</tr>
<tr>
<td>06AF</td>
<td>F1</td>
<td>4886</td>
<td>MOV A. #R1</td>
</tr>
<tr>
<td>06B0</td>
<td>83</td>
<td>4887</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4888</td>
<td>;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4889</td>
<td>;LEBRK LOGICAL FETCH OF BREAK-POINT DATA</td>
</tr>
<tr>
<td>06B1</td>
<td>94E1</td>
<td>4890</td>
<td>LEFBRK: CALL LPSHEL</td>
</tr>
<tr>
<td>06B3</td>
<td>99F7</td>
<td>4891</td>
<td>ANL P1. #NOT 000001000B</td>
</tr>
<tr>
<td>0685</td>
<td>898B</td>
<td>4892</td>
<td>ORL P1. #000001000B</td>
</tr>
<tr>
<td>0687</td>
<td>99FD</td>
<td>4893</td>
<td>ANL P1. #NOT 000000100B</td>
</tr>
<tr>
<td>0689</td>
<td>8981</td>
<td>4894</td>
<td>ORL P1. #00000000001B</td>
</tr>
<tr>
<td>068B</td>
<td>81</td>
<td>4895</td>
<td>MOVX A. #R1</td>
</tr>
<tr>
<td>068C</td>
<td>2381</td>
<td>4896</td>
<td>MOV A. #011H</td>
</tr>
<tr>
<td>068E</td>
<td>86C1</td>
<td>4897</td>
<td>JNI LFEBR1</td>
</tr>
<tr>
<td>068B</td>
<td>27</td>
<td>4898</td>
<td>CLR A</td>
</tr>
<tr>
<td>06C1</td>
<td>83</td>
<td>4899</td>
<td>RET</td>
</tr>
<tr>
<td>004A</td>
<td></td>
<td>4900</td>
<td>SIZECHK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4903</td>
<td>;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4904</td>
<td>;</td>
</tr>
</tbody>
</table>
|      |      | 4905 | ;*******************************
|      |      | 4914 | $EJECT                      |

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0700</td>
<td></td>
<td>=4915</td>
<td>CODEBLK 85</td>
</tr>
<tr>
<td>0700</td>
<td></td>
<td>=4950+</td>
<td>ORG 1792</td>
</tr>
<tr>
<td>0700</td>
<td></td>
<td>=4954 :</td>
<td>LSTORE LOGICAL STORE SUBROUTINE</td>
</tr>
<tr>
<td>0700</td>
<td></td>
<td>=4956 ;</td>
<td>STORES CONTENTS OF LDATA INTO VARIOUS MEMORY SPACES.</td>
</tr>
<tr>
<td>0700</td>
<td></td>
<td>=4957 LSTORE:</td>
<td>MMOV A:TYPE</td>
</tr>
<tr>
<td>0700</td>
<td>B937</td>
<td>=4966+</td>
<td>MOV RI:TYPE</td>
</tr>
<tr>
<td>0702</td>
<td>F1</td>
<td>=4967+</td>
<td>MOV A:ARI</td>
</tr>
<tr>
<td>0703</td>
<td>0996</td>
<td>=4971</td>
<td>ADD A:LOW LSTTEL</td>
</tr>
<tr>
<td>0705</td>
<td>D3</td>
<td>=4972</td>
<td>JMPP</td>
</tr>
<tr>
<td>0706</td>
<td>0C</td>
<td>=4977</td>
<td>LOW LSTPM</td>
</tr>
<tr>
<td>0707</td>
<td>21</td>
<td>=4975</td>
<td>LOW LSTD1M</td>
</tr>
<tr>
<td>0708</td>
<td>26</td>
<td>=4976</td>
<td>LOW LSTREG</td>
</tr>
<tr>
<td>0709</td>
<td>34</td>
<td>=4977</td>
<td>LOW LSTIN</td>
</tr>
<tr>
<td>070A</td>
<td>3D</td>
<td>=4978</td>
<td>LOW LSTOK</td>
</tr>
<tr>
<td>070B</td>
<td>30</td>
<td>=4979</td>
<td>LOW LSTEBK</td>
</tr>
<tr>
<td>070C</td>
<td>B931</td>
<td>=4980</td>
<td>LOW LSTPM:</td>
</tr>
<tr>
<td>070E</td>
<td>F1</td>
<td>=4981</td>
<td>MMOV A:SMALI</td>
</tr>
<tr>
<td>070F</td>
<td>9621</td>
<td>=4982</td>
<td>MOV RI:SMALI</td>
</tr>
<tr>
<td>0711</td>
<td>B930</td>
<td>=4983</td>
<td>MMOV A:SMALO</td>
</tr>
<tr>
<td>0713</td>
<td>F1</td>
<td>=4984+</td>
<td>MOV RI:SMALO</td>
</tr>
<tr>
<td>0714</td>
<td>3E9</td>
<td>=4985</td>
<td>ADD A:O:OVSZC</td>
</tr>
<tr>
<td>0716</td>
<td>F621</td>
<td>=4986</td>
<td>JC LSTD1M</td>
</tr>
<tr>
<td>0718</td>
<td>B930</td>
<td>=4987</td>
<td>MMOV A:SMALO</td>
</tr>
<tr>
<td>071A</td>
<td>F1</td>
<td>=4988</td>
<td>MOV RI:SMALO</td>
</tr>
<tr>
<td>0718</td>
<td>034E</td>
<td>=4989</td>
<td>ADD A:IO:IOBUF</td>
</tr>
<tr>
<td>071D</td>
<td>A9</td>
<td>=4990</td>
<td>MOV RL:A</td>
</tr>
<tr>
<td>071E</td>
<td>FA</td>
<td>=4991</td>
<td>MOV A:DATA</td>
</tr>
<tr>
<td>071F</td>
<td>A1</td>
<td>=4992</td>
<td>MOV ORL:A</td>
</tr>
<tr>
<td>0720</td>
<td>03</td>
<td>=4993</td>
<td>RET</td>
</tr>
<tr>
<td>0721</td>
<td>94E1</td>
<td>=4994</td>
<td>CALL LPSGETK</td>
</tr>
<tr>
<td>0723</td>
<td>FA</td>
<td>=4995</td>
<td>MOV A:DATA</td>
</tr>
<tr>
<td>0724</td>
<td>91</td>
<td>=4996</td>
<td>MOVX ORL:A</td>
</tr>
<tr>
<td>0725</td>
<td>03</td>
<td>=4997</td>
<td>RET</td>
</tr>
<tr>
<td>0726</td>
<td>B939</td>
<td>=4998</td>
<td>LSTREG:</td>
</tr>
<tr>
<td>0728</td>
<td>F1</td>
<td>=4999</td>
<td>MMOV A:SMALO</td>
</tr>
<tr>
<td>0729</td>
<td>557F</td>
<td>=499A</td>
<td>MOV RL:SMALO</td>
</tr>
<tr>
<td>072A</td>
<td>0C2F</td>
<td>=499B</td>
<td>RNL A:R01111111B</td>
</tr>
<tr>
<td>072B</td>
<td>E4C3</td>
<td>=499C</td>
<td>; CHECK IF LOW ORDER BITS = 0</td>
</tr>
<tr>
<td>072F</td>
<td>FA</td>
<td>=499D</td>
<td>LSTREG:</td>
</tr>
<tr>
<td>0733</td>
<td>B923</td>
<td>=499E</td>
<td>MMOV A:DATA</td>
</tr>
<tr>
<td>0732</td>
<td>A1</td>
<td>=499F</td>
<td>MOV RL:HEP0</td>
</tr>
<tr>
<td>0733</td>
<td>03</td>
<td>=49A0</td>
<td>MOV ORL:A</td>
</tr>
<tr>
<td>0733</td>
<td>83</td>
<td>=49A1</td>
<td>RET</td>
</tr>
<tr>
<td>0733</td>
<td>89</td>
<td>=49A2</td>
<td>RET</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ  |  LINE  |  SOURCE STATEMENT
---|---|---
0734 B930  |  =5899+  |  MOV  RL. #SMAHO
0736 F1    |  =5100+  |  MOV  A. #R1
0737 0320  |  =5104  |  ADD  A. #EFACC
0739 A9    |  =5105  |  MOV  RL. A
073A FA    |  =5106  |  MOV  A. #DATA
073B A1    |  =5107  |  MOV  A. RL. A
073C 83    |  =5108  |  RET
          |  =5109 ; |  
          |  =5110 ; |  LSTBRK LOGICAL STORE OF BREAK-POINT DATA
073D 94E1  |  =5111 |  LSTBRK. CALL  LPSEL
073F FA    |  =5112  |  MOV  A. #DATA
0740 1246  |  =5113  |  JBE  LSTBR1
0742 0981  |  =5114  |  ORL  P1. #00000001B
0744 E448  |  =5115  |  JMP  LSTBR2
0746 99FE  |  =5116  |  ANL  P1. #NOT 00000001B
0748 99F7  |  =5117  |  ANL  P1. #NOT 00001000B
074A 81    |  =5118  |  MOVX  A. #R1
074B 0988  |  =5119  |  ORL  P1. #00001000B
074D 83    |  =5120  |  RET
          |  =5121  |  SIZECHK
004E      |  =5124+ |  SIZE  SET  78
          |  =5125+; |  
          |  =5126+; |  CODEBK 17
04E1      |  =5126  |  ORG  1249
          |  =5127  |  LPGSEL LOGICAL PAGE SELECT.
          |  =5128  |  SETS UP PORT 2 TO ADDRESS APPROPRIATE BYTE OF RAM BLOCK.
          |  =5129  |  LPSEL:  MMOV  A. TYPE
04E1 0937  |  =5171+ |  MOV  RL. #TYPE
04E3 F1    |  =5172+ |  MOV  A. #R1
04E4 5301  |  =5176  |  ANL  A. #00000001B ; MASK OFF DATA TYPE SELECTOR BIT
04E6 47    |  =5177  |  SHAP  A
          |  =5178  |  MORA  A. #SKBH
04E7 0931  |  =5104+ |  MOV  RL. #SMAH1
04E9 41    |  =5105+ |  ORL  A. #R1
04EH 4340  |  =5109  |  ORL  A. #0100000B
04EC 3A    |  =5190  |  OUTL  P2. A
          |  =5191  |  MMOV  A. #SMAHO
04ED 0930  |  =5200+ |  MOV  RL. #SMAHO
04EF F1    |  =5201+ |  MOV  A. #R1
04F0 A9    |  =5205  |  MOV  RL. A
04F1 83    |  =5206  |  RET
          |  =5207  |  SIZECHK
0011      |  =5210+ |  SIZE  SET  17
          |  =5211+; |  
          |  =5212+; |  CODEBK 17
          |  =5221 ; |  
          |  =5222 $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE SOURCE STATEMENT

01F2 =5223 CODEBLK 11
   =5223> ORG 498
   =5227 ; INCMA INCREMENT STARTING MEMORY ADDRESS WORD.
01F2 B930 =5238 INCMA: MOV RL @SMALO
01F4 11 =5229 INCH: INC RI
01F5 F1 =5240 MOV A @OR1
01F6 96FC =5241 JNZ INCH1
01F8 19 =5242 INC RI
01F9 F1 =5243 MOV A @OR1
01FA 17 =5244 INC A
01FB 31 =5245 XCHD A @OR1
01FC 03 =5246 INCH1: RET
   =5247 SIZECHK
000C =5250+ SIZE SET 11
   =5251+
   =5252;******************************************************************************
   =5253
   =5254 CODEBLK 12
   =5257+ ORG '756
   =5261 ; DECSIIA DECREMENT SIIA WORD.
02F4 B930 =5262 DECSIIA: MOV RL @SMALO
02F6 F1 =5263 MOV A @OR1
02F7 07 =5264 DEC A
02F8 21 =5265 XCH A @OR1
02F9 96FT =5266 JNZ DECSIIA
02FA 19 =5267 INC RI
02FC F1 =5268 MOV A @OR1
02FD 07 =5269 DEC A
02FE 31 =5270 XCHD A @OR1
02FF 83 =5271 DECSIIA: RET
   =5272 SIZECHK
000C =5275+ SIZE SET 12
   =5276+
   =5277;******************************************************************************
   =5278
   =5279 CODEBLK 15
   =527F
   =5307 ORG '756
   =5314 ; CMPSM COMPARE MEMORY ADDRESSES
   =5317 ; COMPARE SIIA BYTES WITH EMA BYTES TO DETERMINE RELATIVE MAGNITUDE.
   =5319 ; RETURNS WITH CARRY=1 IFF (SMA) => (EMA).
   =531B ; IS CALLED AFTER ACTION HAS BEEN PERFORMED ON (SMA) TO DETERMINE IF
   =531D ; TASK IS COMPLETED:
   =531F ; IF CV=0 THEN (SMA) => (EMA) ==> TERMINATE TASK
   =5321 ; IF CV=1 THEN (SMA) < (EMA) ==> INC SIIA AND REPEAT.
   =5323 CMPSM: MOV A @SMALO
05E2 B930 =5324
   =5325+ MOV RL @SMALO
05E4 F1 =5326+ MOV A @OR1
05E5 37 =5328 CPL A
   =532A MADD A @EMALO
05E6 B932 =532C+ MOV RL @EMALO
05E8 61 =532D+ ADD A @OR1
   =532F+ ; MMOV A @SMARH
05E9 B931 =5331+ MOV RL @SMARH
05EA F1 =5333+ MOV A @OR1
05EC 37 =5335 CPL A

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>5384</td>
<td></td>
<td></td>
<td>ADDOC A.EMPHI</td>
</tr>
<tr>
<td>5390+</td>
<td></td>
<td>MOV</td>
<td>R1.EMPH1</td>
</tr>
<tr>
<td>5391+</td>
<td></td>
<td>ADDOC</td>
<td>A.EMPHI</td>
</tr>
<tr>
<td>5395</td>
<td></td>
<td>CMFRET: RET</td>
<td></td>
</tr>
<tr>
<td>5396</td>
<td></td>
<td>SIZECHK</td>
<td></td>
</tr>
<tr>
<td>5399+</td>
<td></td>
<td>SIZE SET 15</td>
<td></td>
</tr>
<tr>
<td>5400+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5401+</td>
<td></td>
<td>***********************************************</td>
<td></td>
</tr>
<tr>
<td>5410</td>
<td></td>
<td>#EJECT</td>
<td></td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC 0000  LINE  SOURCE STATEMENT

  5411 $  INCLUDE(.F8:KBD.MOD)
=5412 ;   CODEBLK 100
=5413 ;
  074E  =5447+  ORG  1078
=5451 ;
  074F  =5452 ;  KEYBOARD AND DISPLAY PROCESSING ROUTINE.
=5453 ;  CALLED PERIODICALLY WHEN KBD AND DISPLAY ARE TO BE ALIVE.
  074E  O5  =5454 TINT:  SEL  RBI
=5455 ;  MMOV  ASAVE,A
  0751  39  =5468+  MMOV  RL,ASAVE
  0751  A1  =5469+  MMOV  @RL,A
  0752  23F0  =5473 ;  MOV  A,-(-10H)
  0754  62  =5474 ;  MOV  T,A  ;  RELOAD TIMER INTERVAL
  0755  27  =5475 ;  CLR  A
  0756  3E  =5476  ;  MOVD  PSEGHI,A  ;  WRITE BLANK PATTERN TO SEG DRIVERS
  0757  3D  =5477  ;  MOVD  PSEGO,A
  0758  FD  =5478  ;  MOVD  R,CURDIG
  0759  07  =5479 ;  DEC  A
  0759  3F  =5480  ;  MOVD  PDIGIT,A  ;  ENERGIZE CHARACTER
  075A  OC  =5481  ;  MOVD  A,PINPUT  ;  LOAD ANY SWITCH CLOSURES
  075C  AA  =5482 ;  MOV  ROTPAT,A
=5483 ;  WRITE NEXT SEGMENT PATTERN
  075D  FD  =5484 ;  MOV  R,CURDIG
  075E  07  =5485 ;  DEC  A
  075F  0346  =5486  ;  ADD  A,SEGMAP  ;  ADD CURDIG DISPLACEMENT TO BASE
  0761  86  =5487  ;  MOV  RO,A
  0762  F8  =5488  ;  MOV  A,@RO  ;  LOAD ACC W/ NEXT SEGMENT PATTERN
  0763  3D  =5489  ;  MOVD  PSEGO,A  ;  ENABLE APPROPRIATE SEGMENTS
  0764  47  =5490  ;  SNAP  A
  0765  3E  =5491  ;  MOVD  PSEGHI,A
=5492 ;
  0766  B804  =5493 ;  **********************************************
  0766  B804  =5494 ;  THE NEXT CHARACTER IS NOW BEING DISPLAYED.
  0766  B804  =5495 ;  THE KEYBOARD SCAN ROUTINE IS INTEGRATED INTO THE DISPLAY SCAN.
  0766  B804  =5496 ;  WITH THE CURRENT ROW ENERGIZED, CHECK IF THERE ARE ANY INPUTS.
  0766  B804  =5497 ;  **********************************************
  0766  B804  =5498 ;
  0766  B804  =5499 ;  ROTATE BITS THROUGH THE CY WHILE INCREMENTING KEYLOC.
  0766  B804  =5500 ;
  0766  B804  =5501 ;  MOV  ROTONT,NCOL$  ;  SET UP FOR (NCOLS) LOOPS THROUGH 'NEXTLOC'
  0766  B804  =5502  ;  NEXTLOC:  MRRC  ROTPAT
  0767  FA  =5514+  ;  MOV  A,ROPAT
  0767  GF  =5518+  ;  RRC  A
  076A AA  =5529+  ;  MOV  ROTPAT,A
  076B F68B  =5532  ;  JC  SCANS  ;  ONE BIT IN CY INDICATES KEY NOT DOWN
  076C  00E1  =5533  ;  MOV  KEYPAD,#1  ;  MARK THAT AT LEAST ONE KEY WAS DETECTED
  076C  00E1  =5534  ;  A IN THE CURRENT SCAN
  076C  00E1  =5535 ;
  076C  00E1  =5536 ;  **********************************************
  076C  00E1  =5537 ;  A KEYSTROKE WAS DETECTED FOR THE CURRENT COLUMN. ITS
  076C  00E1  =5538 ;  POSITION IS IN REGISTER KEYLOC. SEE IF SAME KEY SENSED LAST CYCLE.
  076C  00E1  =5539 ;  **********************************************
  076C  00E1  =5540 ;
  076C  00E1  =5541 ;  MMOV  A,KEYLOC
  076D  B93C  =5550+  ;  MOV  RL,KEYLOC
  0771  F1  =5551+  ;  MOV  A,#R1
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0772</td>
<td>0C</td>
<td>=5555</td>
<td>XCH A,#LASTKY</td>
</tr>
<tr>
<td>0773</td>
<td>DC</td>
<td>=5556</td>
<td>XRL A,#LASTKY</td>
</tr>
<tr>
<td>0774</td>
<td>C67C</td>
<td>=5557</td>
<td>JZ SCAN3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5558</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5559</td>
<td>; A DIFFERENT KEY WAS READ ON THIS CYCLE THAN ON THE PREVIOUS CYCLE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5560</td>
<td>; SET NREPTS TO THE DEBOUNCE PARAMETER FOR A NEW COUNTDOWN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5561</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5562</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5563</td>
<td>;********************</td>
</tr>
<tr>
<td>0776</td>
<td>B93D</td>
<td>=5564</td>
<td>MOV R1,#NREPTS</td>
</tr>
<tr>
<td>0778</td>
<td>B106</td>
<td>=5565</td>
<td>MOV R1,#N #6</td>
</tr>
<tr>
<td>077A</td>
<td>E488</td>
<td>=5566</td>
<td>JMP SCAN5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5567</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5568</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5569</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5570</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5571</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5572</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5573</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5574</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5575</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5576</td>
<td>;********************</td>
</tr>
<tr>
<td>077C</td>
<td>B93D</td>
<td>=5584</td>
<td>MOV R1,#NREPTS</td>
</tr>
<tr>
<td>077E</td>
<td>F1</td>
<td>=5585</td>
<td>MOV A,#R1</td>
</tr>
<tr>
<td>077F</td>
<td>C688</td>
<td>=5589</td>
<td>JZ SCAN5</td>
</tr>
<tr>
<td>0781</td>
<td>07</td>
<td>=5590</td>
<td>DEC A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5591</td>
<td>;********************</td>
</tr>
<tr>
<td>0782</td>
<td>B93D</td>
<td>=5594</td>
<td>MOV R1,#NREPTS</td>
</tr>
<tr>
<td>0784</td>
<td>A1</td>
<td>=5595</td>
<td>MOV @R1,A</td>
</tr>
<tr>
<td>0785</td>
<td>9688</td>
<td>=5599</td>
<td>JNZ SCAN5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5601</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5602</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5603</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5604</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5605</td>
<td>;********************</td>
</tr>
<tr>
<td>0789</td>
<td>0A1</td>
<td>=5648</td>
<td>MOV R1,#KBDUF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5649</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5650</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5651</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5652</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5653</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5654</td>
<td>;********************</td>
</tr>
<tr>
<td>0790</td>
<td>093C</td>
<td>=5664</td>
<td>MOV R1,#KEYLOC</td>
</tr>
<tr>
<td>0790</td>
<td>11</td>
<td>=5645</td>
<td>INC #R1</td>
</tr>
<tr>
<td>079E</td>
<td>ED68</td>
<td>=5646</td>
<td>DJNZ ROTONT,#X5LOC</td>
</tr>
<tr>
<td>079F</td>
<td>ED88</td>
<td>=5647</td>
<td>DJNZ CURDIG,TIRET1</td>
</tr>
<tr>
<td>07A2</td>
<td>8088</td>
<td>=5648</td>
<td>MOV CURDIG,#CHARS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5649</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5650</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5651</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5652</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5653</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5654</td>
<td>;********************</td>
</tr>
<tr>
<td>0794</td>
<td>B93C</td>
<td>=5666</td>
<td>MOV R1,#KEYLOC</td>
</tr>
<tr>
<td>0796</td>
<td>B100</td>
<td>=5667</td>
<td>MOV @R1,#ZERO</td>
</tr>
<tr>
<td>0798</td>
<td>FC</td>
<td>=5671</td>
<td>MOV A,#KEYFLG</td>
</tr>
<tr>
<td>0799</td>
<td>963D</td>
<td>=5672</td>
<td>JNZ SCAN8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5673</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5674</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5675</td>
<td>;********************</td>
</tr>
<tr>
<td>079B</td>
<td>BCFT</td>
<td>=5676</td>
<td>MOV LASTKY,#NEG1</td>
</tr>
<tr>
<td>079D</td>
<td>BC00</td>
<td>=5682</td>
<td>SCAN8 MOV KEYFLG, #0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5683</td>
<td>;********************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=5684</td>
<td>;********************</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0790</td>
<td>B93F</td>
<td>5665</td>
<td>KBD/DISP RETURN CODE- RESTORES SYSTEM STATUS.</td>
</tr>
<tr>
<td>0790</td>
<td>B93F</td>
<td>5666</td>
<td>MOV RL #DELAY</td>
</tr>
<tr>
<td>0790</td>
<td>B93F</td>
<td>5667</td>
<td>MOV A,R1</td>
</tr>
<tr>
<td>0790</td>
<td>CG03</td>
<td>5701</td>
<td>JZ TIERT1</td>
</tr>
<tr>
<td>0790</td>
<td>07</td>
<td>5702</td>
<td>DEC A</td>
</tr>
<tr>
<td>0790</td>
<td>B93F</td>
<td>5703</td>
<td>MOV KDELAY,A</td>
</tr>
<tr>
<td>0790</td>
<td>A1</td>
<td>5704</td>
<td>MOV RL #DELAY</td>
</tr>
<tr>
<td>0790</td>
<td>B93E</td>
<td>5705</td>
<td>MOV RL #ASAVE</td>
</tr>
<tr>
<td>0790</td>
<td>F1</td>
<td>5706</td>
<td>MOV A,#R1</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5707</td>
<td>RETR</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5708</td>
<td>TOFFOL TIMER OVERFLOW POLLING SUBROUTINE.</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5709</td>
<td>CALLED REPEATEDLY FROM WHEREVER KBD/DISP MUST BE ALIVE.</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5710</td>
<td>MONITORS THE TIMER OVERFLOW FLAG (TOF) AND CALLS SERVICE</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5711</td>
<td>ROUTINE WHEN APPROPRIATE.</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5712</td>
<td>TUPFOL: JTF TIINT</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5713</td>
<td>RET</td>
</tr>
<tr>
<td>0790</td>
<td>93</td>
<td>5714</td>
<td>SIZCCHK</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5715</td>
<td>SIZE SET 97</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5716</td>
<td>RET</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5717</td>
<td>SIZCCHK</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5718</td>
<td>SIZE SET 97</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5719</td>
<td>RET</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5720</td>
<td>SIZCCHK</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5721</td>
<td>SIZE SET 97</td>
</tr>
<tr>
<td>0061</td>
<td>93</td>
<td>5722</td>
<td>RET</td>
</tr>
</tbody>
</table>
LOC OBJ  
LINE  
SOURCE STATEMENT  

06C2  
=5739+  CODEBLK 17  
=5739+  ORG  1730  
=5739+  
=5774  KBDIN  KEYBOARD INPUT SUBROUTINE.  
=5775  RETURN ONLY AFTER A NEW KEYSTROKE HAS BEEN DETECTED AND DEBOUNCED.  
=5776  VALUE OF KEY POSITION IN SWITCH MATRIX IS  
=5777  RETURNED IN THE ACUMULATOR.  
=5778  DISPLAY CHARACTER NOW ON BLANKS BEFORE RETURNING.  
06C2 DF03  
=5779  KBDIN  MOV  XPCCDC. #3  
06C4 74D1  
=5080  CALL  XPTTEST  
06C6 F4AC  
=5801  KBDII:  CALL  TOFPPOL  
=5802  MAOV  A. KEDUF  
06C8 B23B  
=5811+  MOV  RL. #KEDUF  
06CA F1  
=5812+  MOV  A. #R1  
06CB F206  
=5816  JB7  KBDII  
06CD 27  
=5817  CLR  A  
06CE 8E  
=5818  MOVD  PSEGL. A  
06CF 1D  
=5819  MOVD  PSEGR. A  
06D0 37  
=5820  CPL  A  
06D1 21  
=5821  XCH  A. #R1  
06D2 53  
=5822  RET  
06D3  
=5823  SIZECHK  

0811  
=5826+  SIZE SET 17  
=5027+  
=5028+  
=5029+  
=5030+  
=5031+  
=5032+  
=5033+  
=5034+  
=5035+  
=5036+  
=5037+  
=5038+  
=5039+  
=5040+  
=5041+  
=5042+  
=5043+  
=5044+  
=5045+  
=5046+  
=5047+  
=5048+  
=5049+  
=504A+  
=504B+  
=504C+  
=504D+  
=504E+  
=504F+  
=5050+  
=5051+  
=5052+  
=5053+  
=5054+  
=5055+  
=5056+  
=5057+  
=5058+  
=5059+  
=505A+  
=505B+  
=505C+  
=505D+  
=505E+  
=505F+  
=5060+  
=5061+  
=5062+  
=5063+  
=5064+  
=5065+  
=5066+  
=5067+  
=5068+  
=5069+  
=506A+  
=506B+  
=506C+  
=506D+  
=506E+  
=506F+  
=5070+  
=5071+  
=5072+  
=5073+  
=5074+  
=5075+  
=5076+  
=5077+  
=5078+  
=5079+  
=507A+  
=507B+  
=507C+  
=507D+  
=507E+  
=507F+  
=5080+  
=5081+  
=5082+  
=5083+  
=5084+  
=5085+  
=5086+  
=5087+  
=5088+  
=5089+  
=508A+  
=508B+  
=508C+  
=508D+  
=508E+  
=508F+  
=5090+  
=5091+  
=5092+  
=5093+  
=5094+  
=5095+  
=5096+  
=5097+  
=5098+  
=5099+  
=509A+  
=509B+  
=509C+  
=509D+  
=509E+  
=509F+  
=50A0+  
=50A1+  
=50A2+  
=50A3+  
=50A4+  
=50A5+  
=50A6+  
=50A7+  
=50A8+  
=50A9+  
=50AA+  
=50AB+  
=50AC+  
=50AD+  
=50AE+  
=50AF+  
=50B0+  
=50B1+  
=50B2+  
=50B3+  
=50B4+  
=50B5+  
=50B6+  
=50B7+  
=50B8+  
=50B9+  
=50BA+  
=50BB+  
=50BC+  
=50BD+  
=50BE+  
=50BF+  
=50C0+  
=50C1+  
=50C2+  
=50C3+  
=50C4+  
=50C5+  
=50C6+  
=50C7+  
=50C8+  
=50C9+  
=50CA+  
=50CB+  
=50CC+  
=50CD+  
=50CE+  
=50CF+  
=50D0+  
=50D1+  
=50D2+  
=50D3+  
=50D4+  
=50D5+  
=50D6+  
=50D7+  
=50D8+  
=50D9+  
=50DA+  
=50DB+  
=50DC+  
=50DD+  
=50DE+  
=50DF+  
=50E0+  
=50E1+  
=50E2+  
=50E3+  
=50E4+  
=50E5+  
=50E6+  
=50E7+  
=50E8+  
=50E9+  
=50EA+  
=50EB+  
=50EC+  
=50ED+  
=50EE+  
=50EF+  
=50F0+  
=50F1+  
=50F2+  
=50F3+  
=50F4+  
=50F5+  
=50F6+  
=50F7+  
=50F8+  
=50F9+  
=50FA+  
=50FB+  
=50FC+  
=50FD+  
=50FE+  
=50FF+  

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ | LINE | SOURCE STATEMENT

0600 BF04 =5949 MOV XPCODE, #4
0600 74D1 =5950 CALL XTEST
0600 5051 MMOV A.NEXTPL
06DD B93A =5960+ MOV R1.#NEXTPL
06CF F1 =5961+ MOV A.#R1
06E0 83 =5965 ADD A.#SEGMENT-1
06E2 89 =5966 MOV R1.A
06E3 FE =5967 MOV A.DSPTR
06F0 81 =5968 MOV #R1.A
06EE 093A =5969 MOV #R1,#CHARNO
06E5 D93A =5970 MOV R1.#NEXTPL
06E7 F1 =5971 MOV A.#R1
06E0 87 =5972 DEC A
06E9 81 =5973 MOV #R1.A
06EE 88 =5974 MOV #R1,#CHARNO
06E6 093A =5975 MOV R1.#NEXTPL
06F7 F1 =5976 MOV A.#R1
06F0 87 =5977 DEC A
06F9 81 =5978 MOV #R1.A
06EE 88 =5979 MOV #R1,#CHARNO
0700 893F =5980 DB 00111111B ;SEGMENT PATTERN FOR DIGIT '0'
0700 883B =5981 DB 00000110B ;SEGMENT PATTERN FOR DIGIT '1'
0700 883B =5982 DB 01101111B ;SEGMENT PATTERN FOR DIGIT '2'
0700 883B =5983 DB 01001111B ;SEGMENT PATTERN FOR DIGIT '3'
0700 883B =5984 DB 01001110B ;SEGMENT PATTERN FOR DIGIT '4'
0700 883B =5985 DB 01011101B ;SEGMENT PATTERN FOR DIGIT '5'
0700 883B =5986 DB 01110101B ;SEGMENT PATTERN FOR DIGIT '6'
0700 883B =5987 DB 00000111B ;SEGMENT PATTERN FOR DIGIT '7'
0700 883B =5988 DB 01111111B ;SEGMENT PATTERN FOR DIGIT '8'
0700 883B =5989 DB 01100111B ;SEGMENT PATTERN FOR DIGIT '9'
0700 883B =598A DB 01101111B ;SEGMENT PATTERN FOR DIGIT 'A'
0700 883B =598B DB 01111001B ;SEGMENT PATTERN FOR DIGIT 'B'
0700 883B =598C DB 00110011B ;SEGMENT PATTERN FOR DIGIT 'C'
0700 883B =598D DB 01111011B ;SEGMENT PATTERN FOR DIGIT 'D'
0700 883B =598E DB 01100011B ;SEGMENT PATTERN FOR DIGIT 'E'
0700 883B =598F DB 01110011B ;SEGMENT PATTERN FOR DIGIT 'F'
=6816 SIZECHK
=6819+ SIZE SET 44
=6820+
=6821+******************************************************************************
=6830 ;
=6831 CODEBK 12
=6851+ ORG 1266
=6855+ DELAY SUBROUTINE Waits for the number of complete
=6856 ; DISPLAY SCANS CORRESPONDING TO THE ACC CONTENTS.
=6857 ; USED WITH CRUDE HUMAN INTERFACES- AS WHEN OPERATOR SHOULD SEE
=6858 ; SOME DISPLAY CHANGE WHILE IT IS CHANGING:
=6859 DELAY: MMOV RDelay.A
=6872+ MOV R1.#Delay
=6873+ MOV #R1.A

All mnemonics copyrighted © Intel Corporation 1976.

1-146
LOC OBJ LINE SOURCE STATEMENT

04F5 F4AC =6077 DCLAY1: CALL TOFPOL
=6078 MOV A, RDELAY
04F7 D93F =6087+ MOV R1, #RDELAY
04F9 F1 =6088+ MOV A, R1
04FA 9AF5 =60892 JNZ DCLAY1
04FC 83 =60893 RET
=60894 SIZECHK

0000 =6089+ SIZE SET 11
=608A+;
=608B+;
=608C+;#########################################################
=6187 ;
=6188 ;
=6189 ; CODEBLK 8

079F =6144+ ORG 1967
=6148 ;KBDPOL POLL STATUS OF KEYBOARD INPUT ROUTINE.
=6149 ; RETURN WITH ACC BIT 7 = 0 IF KEYBOARD INPUT HAS BEEN RECEIVED.
079F AF85 =6150 KBDPOL MOV XPCODE, #5
07B1 74D1 =6151 CALL XPTST
=6152 MOV A, KDDBUF
07B3 993B =6161+ MOV R1, #KDDBUF
07D5 F1 =6162+ MOV A, R1
07E6 83 =6166 RET
=6167 SIZECHK

0008 =6170+ SIZE SET 8
=6171+;
=6172+;#########################################################
=6181 $EJECT
LOC O8J  
LINE  
SOURCE STATEMENT  

6182 $  INCLUDE (.F0: LINK MOD)  
=6183  CODEBLK 15  
0787  
=6218+  ORG 1975  
=6222 ;EPSTOR FETCH DATA BYTE FROM EP INTERNAL RAM Addressed BY SMALO.  
=6223 EPSTOR: MMOV A, SMALO  
0787 B930  
=6232+  MOV R1, #SMALO  
0789 F1  
=6233+  MOV R0, #R1  
078A F400  
=6237  CALL EPASS  
078C 2380  
=6238  MOV A, $10000000B  
078E F400  
=6239  CALL EPASS  
0790 F400  
=6240  CALL EPASS  
0792 83  
=6241  RET  
0792 83  
=6242  SIZECK  
000C  
=6245+ SIZE SET 12  
=6246+;  
=6247+; *********************************************************  
=6248+  
07C3  
=6292+  ORG 1987  
=6296 ;EPSTOR STORE DATA IN LDATA IN EP INTERNAL RAM AT $SMALO  
=6297 EPSTOR: MMOV A, LDATA  
07C4 F400  
=6298  CALL EPASS  
=6299  MMOV A, SMALO  
07C6 B930  
=6380+  MOV R1, #SMALO  
07C8 F1  
=6380+  MOV R0, #R1  
07C9 5377  
=6313  ANL A, W0111110  
07CB F400  
=6314  CALL EPASS  
07CD F400  
=6315  CALL EPASS  
07CF 83  
=6316  RET  
07D7 83  
=6317  SIZECK  
000D  
=6320+ SIZE SL1 13  
=6321+;  
=6322+; *********************************************************  
=6323 $EJECT  

All mnemonics copyrighted © Intel Corporation 1976.

1-148
LOC OBJ   LINE  SOURCE STATEMENT
        =6322   ; THE FOLLOWING UTILITIES INVOLVE INTERCHANGES BETWEEN THE MP AND EP.
        =6333
        =6334  CODEBLK 11
        =6369+ ORG 2000
        =6373 ; EPPASS PASSES A SINGLE PARAMETER BYTE TO THE EP THROUGH THE LINK.
        =6374 ; WRITE THE CONTENTS OF THE ACC TO THE LINK;
        =6375 ; RELEASE THE EP;
        =6376 ; READ THE LINK INTO THE ACC;
        =6377 ; RETURN.
0700 89C8  =6378 EPPASS: ORL  P2,#001100000 ; ENABLE LINK WRITES.
0702 91   =6379  MOVX  @R1,A ; WRITE ACC TO LINK.
0703 09FE  =6380  ANL  PI,NOT ENBRAM ; DISABLE BREAKPOINTS.
0705 09F2  =6381  ORL  PI,ENBRnk ; SET TO BREAK ON LINK REFERENCE.
0707 F4DB  =6382  CALL  EPSTEP
0709 01   =6383  MOVX  A,@R1
070A 03   =6384  RET
        =6385  SIZECHK
0008  =6388+ SIZE SET 11
        =6389
        =6390+;******************************************************
        =6391 ;
        =6392  CODEBLK 25
        =6393
070B  =6435+ ORG 2011
        =6439 ; EPSTEP RELEASES EP TO RUN IN PRESENT MODE UNTIL AN ANTICIPATED
        =6440 ; HARDWARE BREAK OCCURS.
        =6441 ; (DUE TO SINGLE STEPPING: LINK UNCODE FETCH, OR LINK DATA FETCH)
        =6442 ; MUST OCCUR WITHIN A FINITE NUMBER OF CYCLES (40 MP CYCLES)
        =6443 ; OR WATCHDOG TIMER WILL ASSUME A COMMUNICATIONS ERROR
        =6444 ; BETWEEN THE MP AND EP.
070B F4F4  =6445 EPSTEP: CALL  EPREL
070D 098A  =6446  MOV  R1,#10
070F 8F1   =6447  EPSTE1: JNI  EPSTE2
0711 E0D4  =6448  DJNZ  R1,EPSTE1
0713 8F10  =6449  ORL  PI,#EPSET
0715 744F  =6450  CALL  EBRK
0717 88DB  =6451  MOV  R6,LOW(OVLBAS+OVSIZE)
0719 744A  =6452  CALL  OVLLOAD
071B 99EF  =6453  ANL  PI,NOT EPSET
071D B9BE  =6454  MOV  LDATA,#8CH
071F 249A  =6455  JMP  PERROR
0721 744F  =6456  EPSTE2: CALL  EBRK
0723 83   =6457  RET
        =6458  SIZECHK
0019  =6461+ SIZE SET 25
        =6462
        =6463+;******************************************************
        =6464 ;
        =6465
        =6466  AP·55A

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE source statement

07F4 =6475 CODEBLK 9
     =6510+ ORG 2036
     =6514 ; EPREL RELEASES EP TO RUN IN PRESENT MODE.
     =6515 ; SEQUENCE IS AS FOLLOWS:
     =6516 ; PUT MEMORY ARRAY IN EP MODE;
     =6517 ; RAISE /SSSTEP;
     =6518 ; RETURN.
     =6519 EPREL: ANL P1, NOT CLRBFF ; CLEAR BREAK F/F.
     =6520 ORL P1, CLRBFF ; RE-ENABLE BREAK F/F.
     =6521 ANL P2, NOT 01000000B ; ENABLE EP CONTROL OF MEM ARRAY
     =6522 ORL P1, 00000100B ; FREE EP TO RUN UNTIL BREAK.
     =6523 RET
     =6524 SIZECHK
     =6527+ SIZE SET 9
     =6529+ ;
     =6530 ;
     =6531 CODEBLK 11
     =6533+
     =6534 ; EPREL: REGAIN CONTROL OF MEMORY ARRAY FROM EP.
     =6535 ; DROP /SSSTEP;
     =6536 ; WAIT 30 USECS.;
     =6537 ; PUT MEMORY ARRAY IN MF MODE;
     =6538 ; RETURN.
     =6539 EPREL: ANL P1, NOT 00001000B ; FREEZE EMULATION PROCESSOR.
     =653A ORL P1, INMODOUT ; SIGNAL EP IS NOT RUNNING USER CODE.
     =653B MOV R1, #5
     =653C DJNZ R1, #
     =653D MOV P2, 00100000B
     =653E SET CONTROL OF MEM ARRAY.
     =653F RET
     =6540 SIZECHK
     =6543+ SIZE SET 11
     =654A+
     =654B+ ;
     =654C+
     =654D ;
     =654E ;
     =654F ;
     =6551 CODEBLK 16
     =6552+
     =6553 ; OVSWP OVERLAY SWAP.
     =6554 ; SWAPS BLOCK OF DATABYTES (USER'S PROGRAM) BETWEEN MF RAM & EP PML
     =6555 D885 OVSWP: MOV R8, #00000000B
     =6556 MOV R1, #00000000B
     =6557 MOV R8, #00000000B
     =6558 OUTL P2, A
     =6559 MOV R1, #00000000B
     =655A MOV R8, #00000000B
     =655B MOV R1, #00000000B
     =655C MOV R8, #00000000B
     =655D MOV R1, #00000000B
     =655E MOV R8, #00000000B
     =655F MOV R1, #00000000B
     =6560 MOV R8, #00000000B
     =6561 MOV R1, #00000000B
     =6562 MOV R8, #00000000B
     =6563 MOV R1, #00000000B
     =6564 MOV R8, #00000000B
     =6565 MOV R1, #00000000B
     =6566 MOV R8, #00000000B
     =6567 MOV R1, #00000000B
     =6568 MOV R8, #00000000B
     =6569 MOV R1, #00000000B
     =656A MOV R8, #00000000B
     =656B MOV R1, #00000000B
     =656C MOV R8, #00000000B
     =656D MOV R1, #00000000B
     =656E MOV R8, #00000000B
     =656F MOV R1, #00000000B
     =6570 MOV R8, #00000000B
     =6571 MOV R1, #00000000B
     =6572+ SIZE SET 16

All mnemonics copyrighted © Intel Corporation 1976.

1-150
LOC 000  LINE  SOURCE STATEMENT

=6673+;  *=6674+;  *=6683 ;  *=6684  CODEBLK 14

036A  =6724+  ORG  874 ;
=6728;  OVLDRD  OVERLAY LOAD;
=6729 ;  MOVES  BLOCK  OF  DATABYTES  (ASSEMBLED  SOURCE)  FROM  PG3  TO  EP  PM.
=6730 ;  TOP  OF  DATA  BLOCK  LOADED  AND  BLOCK  LENGTH  DETERMINED  BY  R8  AND  R1.

036A  BD17  =6731  OVLDRD:  MOV  R1,.00000000
036C  2340  =6732  MOV  R1,00000000
036E  3A  =6733  OUTL  P2.A
036E  C8  =6734  MML01:  DEC  R0
0370  C9  =6735  DEC  R1
0371  F8  =6736  MOV  R0, R1
0372  E3  =6737  MOVF3  R0, R1
0373  91  =6738  MOYY  @R1, A
0374  F9  =6739  MOV  A, R1
0375  966F  =6740  JNZ  MML01
0377  63  =6741  RET
=6742  SIZECK

000E  =6745+  SIZE  SET  14
=6746+;  *=6747+;  *=6756  $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ

LINE SOURCE STATEMENT

=6757 

=6758 ;======================================
=6759 ;

=6760 ; THE REST OF THIS MODULE CONTAINS THE MINI-MONITORS WHICH OVERLAY
=6761 ; THE EMULATION PROCESSOR PROGRAM RAM TO GIVE THE
=6762 ; MASTER PROCESSOR ACCESS TO INTERNAL REGISTERS AND RAM OF THE EP.
=6763 ;

=6764 ;======================================
=6765 ;

=6766 ; DATALNK 22

0378 =6771+

=6772+ ORG 008

=6773 ;

=6776 ; DATALNK TO BREAK EP EXECUTION AND JUMP TO LOCATION 009H.
=6777 ; LOCATION 008H REACHED WITH TOP-OF-STACK = RETURN ADDRESS+2
=6778 ; DUE TO FORCED "CALL" DURING WHICH PC WAS INCREMENTED.
=6779 ; LOC 003H & 007H CALL 009H TO SIMULATE SAME CONDITION
=6780 ; IF BREAK OCCURS DURING INTERRUPT CYCLE.
=6781 ; SOURCE CODE FOR MINI-MONITOR OVERLAYED OVER LOW ORDER PROGRAM RAM.

0378 =6782 

=6783 OVBAS EQU $

0378 =6784 ORG OVBAS
0378 1409 =6785 CALL 005H

0379 00 =6786 NOP

=6787 ;

0378 =6788 ORG OVBAS+005H
0378 1409 =6789 CALL 005H
0379 00 =6790 NOP
0379E 00 =6791 NOP

=6792 ;

037F =6793 ORG OVBAS+007H
037F 1409 =6794 CALL 005H
0381 00 =6795 NOP
0382 00 =6796 NOP
0383 00 =6797 NOP
0384 00 =6798 NOP
0385 00 =6799 NOP
0386 00 =6800 NOP
0387 00 =6801 NOP
0388 00 =6802 NOP
0389 00 =6803 NOP
038A 00 =6804 NOP
038B 00 =6805 NOP

=6806 ;

038C =6807 ORG OVBAS+014H
038C 0409 =6808 JMP 005H

=6809 ;

=6818 SIZE CHK

0016 =6819 SIZE SET 22

=6814+ 

=6815+;================================================================================

=6924 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>03E</td>
<td>=6025</td>
<td>=6030+</td>
<td>DATALNK 22</td>
</tr>
<tr>
<td>03E</td>
<td>=6025</td>
<td>=6034+</td>
<td>ORG 910</td>
</tr>
<tr>
<td>03E</td>
<td>=6025</td>
<td>=6035</td>
<td>OVERLAY TO SAVE STATUS DATA AFTER BREAK</td>
</tr>
<tr>
<td>03E</td>
<td>=6025</td>
<td>=6036</td>
<td>ACC, TIMER/COUNTER, PSW (WITH F1), &amp; RAM LOC 0 PASSED SEQUENTIALLY</td>
</tr>
<tr>
<td>03E</td>
<td>=6025</td>
<td>=6037</td>
<td>10 MP.</td>
</tr>
<tr>
<td>03E</td>
<td>=6025</td>
<td>=6038</td>
<td>SOURCE CODE FOR MINI-MONITOR OVERLAYED OVER LOW ORDER PROGRAM RAM</td>
</tr>
<tr>
<td>03E</td>
<td>=6025</td>
<td>=6039</td>
<td></td>
</tr>
<tr>
<td>03E</td>
<td>=6048</td>
<td>OVCBR5 EQU $</td>
<td></td>
</tr>
<tr>
<td>03E</td>
<td>=6049</td>
<td>OVCBR5</td>
<td></td>
</tr>
<tr>
<td>03E</td>
<td>8400</td>
<td>=6042</td>
<td>JMP 006H</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6043</td>
<td>NOP</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6044</td>
<td></td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6045</td>
<td>OVCBR5+003H</td>
</tr>
<tr>
<td>03E</td>
<td>83</td>
<td>=6046</td>
<td>RET</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6047</td>
<td>NOP</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6048</td>
<td>NOP</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6049</td>
<td>NOP</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6050</td>
<td></td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6051</td>
<td>OVCBR5+087H</td>
</tr>
<tr>
<td>03E</td>
<td>83</td>
<td>=6052</td>
<td>RET</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6053</td>
<td>NOP</td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6054</td>
<td></td>
</tr>
<tr>
<td>03E</td>
<td>00</td>
<td>=6055</td>
<td>OVCBR5+089H</td>
</tr>
<tr>
<td>03E</td>
<td>90</td>
<td>=6056</td>
<td>MOV R0, A</td>
</tr>
<tr>
<td>03E</td>
<td>42</td>
<td>=6057</td>
<td>MOV A, T</td>
</tr>
<tr>
<td>03E</td>
<td>90</td>
<td>=6058</td>
<td>MOVX @R0, A</td>
</tr>
<tr>
<td>03E</td>
<td>C7</td>
<td>=6059</td>
<td>MOV R0, PSW</td>
</tr>
<tr>
<td>03E</td>
<td>76</td>
<td>=6060</td>
<td>JF1 OVCBR1</td>
</tr>
<tr>
<td>03E</td>
<td>5F</td>
<td>=6061</td>
<td>ANL R0, #11110111B</td>
</tr>
<tr>
<td>03E</td>
<td>5F</td>
<td>=6062</td>
<td>OVCBR1 EQU $-(LOW OVCBR5)</td>
</tr>
<tr>
<td>03E</td>
<td>90</td>
<td>=6063</td>
<td>MOVX @R0, A</td>
</tr>
<tr>
<td>03E</td>
<td>C5</td>
<td>=6064</td>
<td>SEL RB0</td>
</tr>
<tr>
<td>03E</td>
<td>F8</td>
<td>=6065</td>
<td>MOV R0, R0</td>
</tr>
<tr>
<td>03E</td>
<td>4409</td>
<td>=6066</td>
<td>JMP 005H</td>
</tr>
<tr>
<td>03E</td>
<td>09</td>
<td>=6067</td>
<td></td>
</tr>
<tr>
<td>03E</td>
<td>09</td>
<td>=6068</td>
<td>SIZECHK</td>
</tr>
<tr>
<td>0016</td>
<td>=6071</td>
<td>=6072+</td>
<td>SIZE SET 22</td>
</tr>
<tr>
<td>0016</td>
<td>=6073</td>
<td>=6074</td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>=6075</td>
<td>=6076</td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>=6077</td>
<td>=6078</td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>=6079</td>
<td>=607A</td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>=607B</td>
<td>=607C</td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>=607D</td>
<td>=607E</td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>=607F</td>
<td>=6080</td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>=6081</td>
<td>=6082</td>
<td>EJECT</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
LOC   OBJ   LINE   SOURCE STATEMENT

03A4 =6883   DATALNK 22
=6884+   ORG 932
=6892 ;
=6893 ; 0V1-  OVERLAY 1 TO GIVE MP ACCESS TO EP RAM LOC5 06H-7FH.
=6894 ;  SOURCE CODE FOR MINI-MONITOR OVERLAYER OVER LOW ORDER PROGRAM RAM.
=6895 ;
03A4 =6896 OVI.BAS EQU $   =6897 ;
03A5 040R  =6898   JMP OVI.B1   =6899   NOP
03A6  00   =6900 ;
03A7 =6901 ORG OVI.BNS+03H
03A8  03   =6902   RET
03A9  00   =6903   NOP
03A9  00   =6904   NOP
03A9  00   =6905   NOP
=6906 ;
03A8 =6907 ORG OVI.BNS+07H
03A9  03   =6908   RET
03A9  00   =6909   NOP
=6910 ;
03A9 =6911 ORG OVI.BNS+09H
03A9  00   =6912   MO VX @R8,A
=6913 ;
000A =6914 OVI.B1 EQU $- OVI.BAS   =6915 ;
03B0  00   =6916   MO VX A,@R8
03B0  00   =6917   MOV @R8,A
03B0  00   =6918   MO VX @R8,A
03B0 F213  =6919   JB7  OVI.B2
03B0  28   =6920   XCH A,@R8
03B1  00   =6921   MOV @R8,A
03B2  0409   =6922   JMP 003H
=6923 ;
03B3 =6924 OVI.B2 EQU $-LOW OVI.BAS   =6925 ;
03B3 F0   =6926   MOV A,@R8
03B3  0409   =6927   JMP 003H
=6928 ;
=6929   SIZECHK
0016 =69324 SIZE SET 22
=6933A ;
>6934#;************************************************************************************************************
>6943 #EJECT

All mnemonics copyrighted © Intel Corporation 1976.
LOC OBJ LINE SOURCE STATEMENT

=6944 DATABLK 23
=6949+ ORG 354
=6951 ;
=6954 : OV2- OVERLAY TO RESTORE EP STATUS SAVED ON BREAK AND RESUME USER'S PROGRAM.
=6955 ; SOURCE CODE FOR MINI-MONITOR OVERLayed OVER LOW ORDER PROGRAM RAM.
=6956 ;

=6957 OV2BRS EQU $ ;
=6958 ORG OV2BRS
=6959 JMP 000H
=6960 NOP
=6961 ;

=6962 ORG OV2BRS+003H
=6963 RET
=6964 NOP
=6965 NOP
=6966 NOP
=6967 ;
=6968 ORG OV2BRS+007H
=6969 RET
=6970 NOP
=6971 ;

=6972 ORG OV2BRS+009H
=6973 MOVX @R0, A
=6974 ;
=6975 MOVX A, @R0
=6976 MOV R0, A
=6977 MOVX A, @R0
=6978 MOV 15H, A
=6979 CLR F1
=6980 CPI F1
=6981 JBC OV2B1
=6982 CLR F1
=6983 ;
=6984 OV2B1 EQU $-LOW OV2BRS
=6985 ;
=6986 MOVX A, @R0
=6987 MOV T, A
=6988 MOVX A, @R0
=6989 RETR
=6990 SIZE DLK
=6991 ;

=6993+ SIZE SET 23
=6994+ ;
=6995+******************************************************************************
=7004 $EJECT

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>7005</td>
<td></td>
<td></td>
<td>CODEBLK 11</td>
</tr>
<tr>
<td>03D1</td>
<td></td>
<td>7006</td>
<td>ORG 977</td>
</tr>
<tr>
<td>03D1</td>
<td>0900</td>
<td>7050</td>
<td>XTEST: ORL P2, #80H</td>
</tr>
<tr>
<td>03D3</td>
<td>0A</td>
<td>7051</td>
<td>IN A, P2</td>
</tr>
<tr>
<td>03D4</td>
<td>97</td>
<td>7052</td>
<td>ANL P2, #(#NOT 08H)</td>
</tr>
<tr>
<td>03D6</td>
<td>F209</td>
<td>7053</td>
<td>JGP #3</td>
</tr>
<tr>
<td>03D8</td>
<td>03</td>
<td>7054</td>
<td>RET</td>
</tr>
<tr>
<td>03D9</td>
<td>05</td>
<td>7055</td>
<td>SCL MBL</td>
</tr>
<tr>
<td>03DA</td>
<td>0400</td>
<td>7056</td>
<td>JMP 000H</td>
</tr>
<tr>
<td>7057</td>
<td></td>
<td></td>
<td>SIZECHK</td>
</tr>
<tr>
<td>0000</td>
<td>7060</td>
<td>SIZE</td>
<td>SET 11</td>
</tr>
<tr>
<td></td>
<td>7061</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|      | 7062 |      | ***************** |}
| 03DC | 7112 | 7071 | CODEBLK 13      |
| 03DC | 20432931 | 7116 | DB '©1979 INTEL' |
| 03EB | 3937920 |      |                 |
| 03E4 | 494E5445 |      |                 |
| 03E8 | 04 | 7117 | SIZECHK        |
| 0000 | 7120 | SIZE | SET 13         |
|      | 7121 |      |                 |
|      | 7122 |      | ***************** |}
<p>| 7131 | ;  |
| 7132 | ;  |
| 7133 | ;  |
| 0100 | 7135 | PGSIZE | SET ORGP0-000H ;BYTES USED ON PAGE 0 |
| 0100 | 7136 | PGSIZE | SET ORGP1-100H ;BYTES USED ON PAGE 1 |
| 0100 | 7137 | PGSIZE | SET ORP02-200H ;BYTES USED ON PAGE 2 |
| 0100 | 7138 | PGSIZE | SET ORP03-300H ;BYTES USED ON PAGE 3 |
| 0100 | 7139 | PGSIZE | SET ORP04-400H ;BYTES USED ON PAGE 4 |
| 0100 | 7140 | PGSIZE | SET ORP05-500H ;BYTES USED ON PAGE 5 |
| 0100 | 7141 | PGSIZE | SET ORP06-600H ;BYTES USED ON PAGE 6 |
| 0100 | 7142 | PGSIZE | SET ORP07-700H ;BYTES USED ON PAGE 7 |
| 7143 | ;  |      |     |</p>
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>7145</td>
<td>;*******************************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7146</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7147</td>
<td>; FILL ALL UNUSED MEMORY LOCATIONS WITH NOP OPCODES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7148</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7149</td>
<td>;*******************************</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7150</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7151</td>
<td>#GEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7152</td>
<td></td>
</tr>
<tr>
<td>81FD</td>
<td></td>
<td>7160</td>
<td>ORG ORGPG1</td>
</tr>
<tr>
<td></td>
<td>7161</td>
<td>REPT (200H - ORGPG1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7162</td>
<td>DB 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7163</td>
<td>ENDM</td>
<td></td>
</tr>
<tr>
<td>81FD 00</td>
<td></td>
<td>7164</td>
<td>DB 0</td>
</tr>
<tr>
<td>81FE 00</td>
<td></td>
<td>7165</td>
<td>DB 0</td>
</tr>
<tr>
<td>81FF 00</td>
<td></td>
<td>7166</td>
<td>DB 0</td>
</tr>
<tr>
<td></td>
<td>7168</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>83E9</td>
<td></td>
<td>7177</td>
<td>ORG ORGPG3</td>
</tr>
<tr>
<td></td>
<td>7178</td>
<td>REPT (400H - ORGPG3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7179</td>
<td>DB 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7180</td>
<td>ENDM</td>
<td></td>
</tr>
<tr>
<td>83E9 00</td>
<td></td>
<td>7181</td>
<td>DB 0</td>
</tr>
<tr>
<td>83EA 00</td>
<td></td>
<td>7182</td>
<td>DB 0</td>
</tr>
<tr>
<td>83EB 00</td>
<td></td>
<td>7183</td>
<td>DB 0</td>
</tr>
<tr>
<td>83EC 00</td>
<td></td>
<td>7184</td>
<td>DB 0</td>
</tr>
<tr>
<td>83ED 00</td>
<td></td>
<td>7185</td>
<td>DB 0</td>
</tr>
<tr>
<td>83EE 00</td>
<td></td>
<td>7186</td>
<td>DB 0</td>
</tr>
<tr>
<td>83EF 00</td>
<td></td>
<td>7187</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F0 00</td>
<td></td>
<td>7188</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F1 00</td>
<td></td>
<td>7189</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F2 00</td>
<td></td>
<td>7190</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F3 00</td>
<td></td>
<td>7191</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F4 00</td>
<td></td>
<td>7192</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F5 00</td>
<td></td>
<td>7193</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F6 00</td>
<td></td>
<td>7194</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F7 00</td>
<td></td>
<td>7195</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F8 00</td>
<td></td>
<td>7196</td>
<td>DB 0</td>
</tr>
<tr>
<td>03F9 00</td>
<td></td>
<td>7197</td>
<td>DB 0</td>
</tr>
<tr>
<td>03FA 00</td>
<td></td>
<td>7198</td>
<td>DB 0</td>
</tr>
<tr>
<td>03FB 00</td>
<td></td>
<td>7199</td>
<td>DB 0</td>
</tr>
<tr>
<td>03FC 00</td>
<td></td>
<td>7200</td>
<td>DB 0</td>
</tr>
<tr>
<td>03FD 00</td>
<td></td>
<td>7201</td>
<td>DB 0</td>
</tr>
<tr>
<td>03FE 00</td>
<td></td>
<td>7202</td>
<td>DB 0</td>
</tr>
<tr>
<td>03FF 00</td>
<td></td>
<td>7203</td>
<td>DB 0</td>
</tr>
<tr>
<td></td>
<td>7205</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>04FD</td>
<td></td>
<td>7207</td>
<td>ORG ORGPG4</td>
</tr>
<tr>
<td></td>
<td>7208</td>
<td>REPT (500H - ORGPG4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7209</td>
<td>DB 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7210</td>
<td>ENDM</td>
<td></td>
</tr>
<tr>
<td>04FD 00</td>
<td></td>
<td>7211</td>
<td>DB 0</td>
</tr>
<tr>
<td>04FE 00</td>
<td></td>
<td>7212</td>
<td>DB 0</td>
</tr>
<tr>
<td>04FF 00</td>
<td></td>
<td>7213</td>
<td>DB 0</td>
</tr>
<tr>
<td></td>
<td>7215</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>05FF</td>
<td></td>
<td>7217</td>
<td>ORG ORGPG5</td>
</tr>
<tr>
<td></td>
<td>7218</td>
<td>REPT (600H - ORGPG5)</td>
<td></td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>LOC 0CJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>7219</td>
<td>DC</td>
<td>0</td>
</tr>
<tr>
<td>7220</td>
<td>ENDM</td>
<td></td>
</tr>
<tr>
<td>05FF 00</td>
<td>DB</td>
<td>0</td>
</tr>
<tr>
<td>7223</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06FF</td>
<td>ORG</td>
<td>ORGPG6</td>
</tr>
<tr>
<td>7225</td>
<td>REPT</td>
<td>(700H - ORGPG6)</td>
</tr>
<tr>
<td>7226</td>
<td>DB</td>
<td>0</td>
</tr>
<tr>
<td>7227</td>
<td>ENDM</td>
<td></td>
</tr>
<tr>
<td>06FF 00</td>
<td>DB</td>
<td>0</td>
</tr>
<tr>
<td>7229</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7231</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07FD</td>
<td>ORG</td>
<td>ORGPG7</td>
</tr>
<tr>
<td>7233</td>
<td>REPT</td>
<td>(800H - ORGPG7)</td>
</tr>
<tr>
<td>7234</td>
<td>DB</td>
<td>0</td>
</tr>
<tr>
<td>7235</td>
<td>ENDM</td>
<td></td>
</tr>
<tr>
<td>07FD 00</td>
<td>DB</td>
<td>0</td>
</tr>
<tr>
<td>07TE 00</td>
<td>DB</td>
<td>0</td>
</tr>
<tr>
<td>07FF 00</td>
<td>DB</td>
<td>0</td>
</tr>
<tr>
<td>7241</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7242</td>
<td>$EJECT</td>
<td></td>
</tr>
<tr>
<td>LFILL</td>
<td>4670</td>
<td>4676</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>LFILL</td>
<td>4674</td>
<td>4677</td>
</tr>
<tr>
<td>LPOSET</td>
<td>4832</td>
<td>4830</td>
</tr>
<tr>
<td>LSTBLR1</td>
<td>5113</td>
<td>5116</td>
</tr>
<tr>
<td>LSTBLR2</td>
<td>5115</td>
<td>5117</td>
</tr>
<tr>
<td>LSTBLR</td>
<td>5178</td>
<td>5179</td>
</tr>
<tr>
<td>LSTDM</td>
<td>4975</td>
<td>4995</td>
</tr>
<tr>
<td>LSTINT</td>
<td>4977</td>
<td>5090</td>
</tr>
<tr>
<td>LSTORE</td>
<td>2459</td>
<td>2815</td>
</tr>
<tr>
<td>LSTMP</td>
<td>4974</td>
<td>4981</td>
</tr>
<tr>
<td>LSTORE</td>
<td>5665</td>
<td>5665</td>
</tr>
<tr>
<td>LSTREG</td>
<td>4976</td>
<td>5037</td>
</tr>
<tr>
<td>LSTBL</td>
<td>4971</td>
<td>4974</td>
</tr>
<tr>
<td>M0</td>
<td>5510</td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>5520</td>
<td></td>
</tr>
<tr>
<td>MADDO</td>
<td>4380</td>
<td>1816</td>
</tr>
<tr>
<td>MADDC</td>
<td>4350</td>
<td>5984</td>
</tr>
<tr>
<td>MAIN</td>
<td>1434</td>
<td>1539</td>
</tr>
<tr>
<td>MAIN2</td>
<td>1544</td>
<td>3129</td>
</tr>
<tr>
<td>MAINR</td>
<td>1594</td>
<td>1699</td>
</tr>
<tr>
<td>MAINR</td>
<td>1672</td>
<td>1674</td>
</tr>
<tr>
<td>MAINBR</td>
<td>1798</td>
<td>1328</td>
</tr>
<tr>
<td>MAINBD</td>
<td>1831</td>
<td>1847</td>
</tr>
<tr>
<td>MAINC</td>
<td>1716</td>
<td>1762</td>
</tr>
<tr>
<td>MAIND</td>
<td>1741</td>
<td>1381</td>
</tr>
<tr>
<td>MAINB</td>
<td>1742</td>
<td>1766</td>
</tr>
<tr>
<td>MAUL</td>
<td>4480</td>
<td></td>
</tr>
<tr>
<td>MAUL</td>
<td>1550</td>
<td>1387</td>
</tr>
<tr>
<td>MDEC</td>
<td>4710</td>
<td>5929</td>
</tr>
<tr>
<td>MDNZ</td>
<td>4750</td>
<td>4041</td>
</tr>
<tr>
<td>MERHI</td>
<td>11580</td>
<td>3824</td>
</tr>
<tr>
<td>MERM</td>
<td>11490</td>
<td>3851</td>
</tr>
<tr>
<td>MERR</td>
<td>1520</td>
<td>1058</td>
</tr>
<tr>
<td>NINC</td>
<td>4670</td>
<td>1743</td>
</tr>
<tr>
<td>MMUL</td>
<td>6730</td>
<td>6740</td>
</tr>
<tr>
<td>MMUL</td>
<td>3950</td>
<td>1558</td>
</tr>
<tr>
<td>2464</td>
<td>2482</td>
<td>2541</td>
</tr>
<tr>
<td>3002</td>
<td>3061</td>
<td>3091</td>
</tr>
<tr>
<td>3518</td>
<td>3557</td>
<td>2579</td>
</tr>
<tr>
<td>4639</td>
<td>4739</td>
<td>4793</td>
</tr>
<tr>
<td>5191</td>
<td>5343</td>
<td>5369</td>
</tr>
<tr>
<td>6959</td>
<td>6878</td>
<td>6152</td>
</tr>
<tr>
<td>MODOUT</td>
<td>5570</td>
<td>3041</td>
</tr>
<tr>
<td>MORD</td>
<td>4450</td>
<td>2988</td>
</tr>
<tr>
<td>MFUSEL</td>
<td>5530</td>
<td></td>
</tr>
<tr>
<td>MR</td>
<td>4820</td>
<td></td>
</tr>
<tr>
<td>MRC</td>
<td>4940</td>
<td></td>
</tr>
<tr>
<td>MRR</td>
<td>4860</td>
<td></td>
</tr>
<tr>
<td>MRC</td>
<td>4980</td>
<td>4437</td>
</tr>
<tr>
<td>MDCH</td>
<td>4550</td>
<td></td>
</tr>
<tr>
<td>MOL</td>
<td>4580</td>
<td></td>
</tr>
<tr>
<td>MCOLS</td>
<td>6140</td>
<td>5581</td>
</tr>
<tr>
<td>NEG</td>
<td>7290</td>
<td>2924</td>
</tr>
<tr>
<td>NEXTPL</td>
<td>1280</td>
<td>2253</td>
</tr>
<tr>
<td>NI</td>
<td>3702</td>
<td>3700</td>
</tr>
<tr>
<td>NILIN</td>
<td>3627</td>
<td>3638</td>
</tr>
<tr>
<td>NIBING</td>
<td>3953</td>
<td>3700</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Decimal Value</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS3</td>
<td>7141</td>
<td>01110001001</td>
</tr>
<tr>
<td>TTYOUT</td>
<td>539#</td>
<td>4428</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>TYPE</td>
<td>1176#</td>
<td>1429</td>
</tr>
<tr>
<td>UPDHDL</td>
<td>2205#</td>
<td>2558</td>
</tr>
<tr>
<td>UPDADR</td>
<td>2195</td>
<td>2249#</td>
</tr>
<tr>
<td>VERSNO</td>
<td>1059#</td>
<td></td>
</tr>
<tr>
<td>WB4X</td>
<td>1522#</td>
<td>1928</td>
</tr>
<tr>
<td>WDISP</td>
<td>2010</td>
<td>2030</td>
</tr>
<tr>
<td>WDISP1</td>
<td>9998</td>
<td>999#</td>
</tr>
<tr>
<td>XPCODE</td>
<td>837#</td>
<td>1410</td>
</tr>
<tr>
<td>XFTEST</td>
<td>1411</td>
<td>1540</td>
</tr>
<tr>
<td>ZERO</td>
<td>694#</td>
<td>1570</td>
</tr>
</tbody>
</table>

CROSS REFERENCE COMPLETE
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRKFIL 1266</td>
<td>2433</td>
</tr>
<tr>
<td>BRKNEXT 2439</td>
<td>2499</td>
</tr>
<tr>
<td>BUFCT 3534</td>
<td>3534</td>
</tr>
<tr>
<td>BUFLEN 1329</td>
<td>3875</td>
</tr>
<tr>
<td>BYTEI 3554</td>
<td>3220</td>
</tr>
<tr>
<td>BYTEIN 3432</td>
<td>3489</td>
</tr>
<tr>
<td>BYTED 3995</td>
<td>4025</td>
</tr>
<tr>
<td>COO 2986</td>
<td>3002</td>
</tr>
<tr>
<td>COOBR 3819</td>
<td>3020</td>
</tr>
<tr>
<td>COOPAT 3822</td>
<td>3025</td>
</tr>
<tr>
<td>COSS 3821</td>
<td>3034</td>
</tr>
<tr>
<td>COUADD 3823</td>
<td>3033</td>
</tr>
<tr>
<td>COOBR 3820</td>
<td>3026</td>
</tr>
<tr>
<td>CHACR 3933</td>
<td>4119</td>
</tr>
<tr>
<td>CHARIN 3417</td>
<td>3549</td>
</tr>
<tr>
<td>CHARLF 3394</td>
<td>4121</td>
</tr>
<tr>
<td>CHARNO 589</td>
<td>1349</td>
</tr>
<tr>
<td>CHARP 3791</td>
<td>3977</td>
</tr>
<tr>
<td>CHKERR 3577</td>
<td>3598</td>
</tr>
<tr>
<td>CHKSUM 2831</td>
<td>3566</td>
</tr>
<tr>
<td>CIL 4514</td>
<td>4531</td>
</tr>
<tr>
<td>CILL 4533</td>
<td>4534</td>
</tr>
<tr>
<td>CILS 4537</td>
<td>4556</td>
</tr>
<tr>
<td>CINE 3749</td>
<td>4529</td>
</tr>
<tr>
<td>CKSMOK 3551</td>
<td>3578</td>
</tr>
<tr>
<td>CLEAR 1974</td>
<td>5078</td>
</tr>
<tr>
<td>CLRFST 534</td>
<td>6529</td>
</tr>
<tr>
<td>CMDINT 1836</td>
<td>1845</td>
</tr>
<tr>
<td>CMPARR 3871</td>
<td>4673</td>
</tr>
<tr>
<td>CMPPTR 5395</td>
<td>5343</td>
</tr>
<tr>
<td>CTNLZ 3395</td>
<td>3418</td>
</tr>
<tr>
<td>CNTTBL 3877</td>
<td>3880</td>
</tr>
<tr>
<td>CNTTRR 3881</td>
<td>3891</td>
</tr>
<tr>
<td>COO 4427</td>
<td>4475</td>
</tr>
<tr>
<td>CO2 4427</td>
<td>4438</td>
</tr>
<tr>
<td>CO4 4429</td>
<td>4433</td>
</tr>
<tr>
<td>CODECL 1596</td>
<td>1945</td>
</tr>
<tr>
<td>CODELT 1596</td>
<td>2155</td>
</tr>
<tr>
<td>COMBCR 2407</td>
<td>2432</td>
</tr>
<tr>
<td>COMFIL 1428</td>
<td>2426</td>
</tr>
<tr>
<td>COMGCR 2429</td>
<td>2992</td>
</tr>
<tr>
<td>COMGCR 2429</td>
<td>2992</td>
</tr>
<tr>
<td>COMSIZ 1596</td>
<td>1899</td>
</tr>
<tr>
<td>CRAB 1557</td>
<td>1898</td>
</tr>
<tr>
<td>CURDIG 3200</td>
<td>5464</td>
</tr>
<tr>
<td>DATABR 2440</td>
<td>1332</td>
</tr>
<tr>
<td>DATO 4829</td>
<td>4833</td>
</tr>
<tr>
<td>DATOI 4834</td>
<td>4860</td>
</tr>
<tr>
<td>DELLK 2215</td>
<td>5872</td>
</tr>
<tr>
<td>DBPNT 2636</td>
<td>2085</td>
</tr>
<tr>
<td>DRRK 1519</td>
<td>1924</td>
</tr>
<tr>
<td>DUC 2045</td>
<td>2196</td>
</tr>
<tr>
<td>DURK 2653</td>
<td>2122</td>
</tr>
<tr>
<td>DURMK 2849</td>
<td>2116</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>BLITCE</th>
<th>538#</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLAR</td>
<td>1798</td>
</tr>
<tr>
<td></td>
<td>848</td>
</tr>
<tr>
<td></td>
<td>1063</td>
</tr>
<tr>
<td></td>
<td>1207</td>
</tr>
<tr>
<td>DECSEM</td>
<td>5286</td>
</tr>
<tr>
<td>DECSEM</td>
<td>5292#</td>
</tr>
<tr>
<td>DELAY</td>
<td>3185</td>
</tr>
<tr>
<td>DELAY1</td>
<td>5277</td>
</tr>
<tr>
<td>DERROR</td>
<td>2033</td>
</tr>
<tr>
<td>DFILL</td>
<td>2848</td>
</tr>
<tr>
<td>DGO</td>
<td>2839</td>
</tr>
<tr>
<td>DGR</td>
<td>2846</td>
</tr>
<tr>
<td>DINTRO</td>
<td>2851</td>
</tr>
<tr>
<td>DLST</td>
<td>2841</td>
</tr>
<tr>
<td>DNOBKR</td>
<td>2855</td>
</tr>
<tr>
<td>DOME</td>
<td>3419</td>
</tr>
<tr>
<td>DPA</td>
<td>2858</td>
</tr>
<tr>
<td>DPREBKR</td>
<td>2852</td>
</tr>
<tr>
<td>DFPMEM</td>
<td>2846</td>
</tr>
<tr>
<td>DREC</td>
<td>2842</td>
</tr>
<tr>
<td>DREL</td>
<td>2843</td>
</tr>
<tr>
<td>DRM</td>
<td>2850</td>
</tr>
<tr>
<td>DRUN</td>
<td>2835</td>
</tr>
<tr>
<td>DSG</td>
<td>2844</td>
</tr>
<tr>
<td>DSIGNON</td>
<td>2834</td>
</tr>
<tr>
<td>DSHACC</td>
<td>2276</td>
</tr>
<tr>
<td>DSH1</td>
<td>2260</td>
</tr>
<tr>
<td>DSHO</td>
<td>2275</td>
</tr>
<tr>
<td>DSHM</td>
<td>2272</td>
</tr>
<tr>
<td>DSHMD</td>
<td>2277#</td>
</tr>
<tr>
<td>DSPTIM</td>
<td>1041#</td>
</tr>
<tr>
<td>DSTM#</td>
<td>3280</td>
</tr>
<tr>
<td>DSS</td>
<td>2857</td>
</tr>
<tr>
<td>DTR</td>
<td>2859</td>
</tr>
<tr>
<td>DWDKR</td>
<td>2856</td>
</tr>
<tr>
<td>ELSTF1</td>
<td>2186</td>
</tr>
<tr>
<td>ELSTF2</td>
<td>2204</td>
</tr>
<tr>
<td>EMWH1</td>
<td>1140#</td>
</tr>
<tr>
<td>EMW0</td>
<td>1131#</td>
</tr>
<tr>
<td>ENBLK</td>
<td>5291</td>
</tr>
<tr>
<td>ENBRAM</td>
<td>5290</td>
</tr>
<tr>
<td>ENDF1</td>
<td>3888#</td>
</tr>
<tr>
<td>ENDFIL</td>
<td>3872</td>
</tr>
<tr>
<td>ENDREC</td>
<td>4864#</td>
</tr>
<tr>
<td>EOFREC</td>
<td>3887</td>
</tr>
<tr>
<td>EFYCC</td>
<td>969#</td>
</tr>
<tr>
<td>EPBKR</td>
<td>1425</td>
</tr>
<tr>
<td>EPCONT</td>
<td>2921#</td>
</tr>
<tr>
<td>EPCON1</td>
<td>2785</td>
</tr>
<tr>
<td>EPCONT</td>
<td>2780</td>
</tr>
<tr>
<td>EFEKTE</td>
<td>3319</td>
</tr>
<tr>
<td>EPPRS1</td>
<td>2937</td>
</tr>
<tr>
<td>EPPCHI</td>
<td>1014#</td>
</tr>
<tr>
<td>EPPCH10</td>
<td>1005#</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPPSN</td>
<td>2890</td>
</tr>
<tr>
<td>EPR0</td>
<td>2932</td>
</tr>
<tr>
<td>EPR1</td>
<td>2978</td>
</tr>
<tr>
<td>EPREL</td>
<td>3042</td>
</tr>
<tr>
<td>EPRSET</td>
<td>3110</td>
</tr>
<tr>
<td>EPRSET1</td>
<td>5364</td>
</tr>
<tr>
<td>EPR0UN</td>
<td>2424</td>
</tr>
<tr>
<td>EPR0UN1</td>
<td>3840</td>
</tr>
<tr>
<td>EPR0UN2</td>
<td>3850</td>
</tr>
<tr>
<td>EPR0UN3</td>
<td>3849</td>
</tr>
<tr>
<td>EPR0UN4</td>
<td>3828</td>
</tr>
<tr>
<td>EPR0UN5</td>
<td>3857</td>
</tr>
<tr>
<td>EPR0UN6</td>
<td>3881</td>
</tr>
<tr>
<td>EPSSTP</td>
<td>3322</td>
</tr>
<tr>
<td>EPSST1</td>
<td>6447</td>
</tr>
<tr>
<td>EPSST2</td>
<td>6447</td>
</tr>
<tr>
<td>EPSSTEP</td>
<td>2904</td>
</tr>
<tr>
<td>EPSSTOR</td>
<td>2574</td>
</tr>
<tr>
<td>EPTMPR</td>
<td>5872</td>
</tr>
<tr>
<td>ERROR1</td>
<td>740</td>
</tr>
<tr>
<td>ERROR2</td>
<td>2324</td>
</tr>
<tr>
<td>EXMN1</td>
<td>2541</td>
</tr>
<tr>
<td>EXMN2</td>
<td>2619</td>
</tr>
<tr>
<td>EXMN3</td>
<td>2624</td>
</tr>
<tr>
<td>EXMN4</td>
<td>2629</td>
</tr>
<tr>
<td>EXMN5</td>
<td>2610</td>
</tr>
<tr>
<td>EXMN6</td>
<td>2410</td>
</tr>
<tr>
<td>EXPMON</td>
<td>5555</td>
</tr>
<tr>
<td>FDUMP1</td>
<td>3978</td>
</tr>
<tr>
<td>FDUMP2</td>
<td>4026</td>
</tr>
<tr>
<td>FDUMP3</td>
<td>4035</td>
</tr>
<tr>
<td>FDUMP4</td>
<td>4064</td>
</tr>
<tr>
<td>FDUMP5</td>
<td>4034</td>
</tr>
<tr>
<td>FINDIR1</td>
<td>1598</td>
</tr>
<tr>
<td>GOTBL1</td>
<td>3016</td>
</tr>
<tr>
<td>HBD1</td>
<td>4317</td>
</tr>
<tr>
<td>HBD2</td>
<td>4310</td>
</tr>
<tr>
<td>HBDH'</td>
<td>4257</td>
</tr>
<tr>
<td>HEITH1</td>
<td>1832</td>
</tr>
<tr>
<td>HETILO</td>
<td>1823</td>
</tr>
<tr>
<td>HDATIN</td>
<td>3541</td>
</tr>
<tr>
<td>HEXRSC1</td>
<td>4193</td>
</tr>
<tr>
<td>HEXBUF1</td>
<td>1217</td>
</tr>
<tr>
<td>HEXVIB1</td>
<td>4135</td>
</tr>
<tr>
<td>HFDONE1</td>
<td>3898</td>
</tr>
<tr>
<td>HFLEO1</td>
<td>2413</td>
</tr>
<tr>
<td>HRECIN1</td>
<td>2416</td>
</tr>
<tr>
<td>HRECO1</td>
<td>3877</td>
</tr>
<tr>
<td>HRECO2</td>
<td>1859</td>
</tr>
<tr>
<td>HRECO3</td>
<td>1869</td>
</tr>
<tr>
<td>HRECO4</td>
<td>1869</td>
</tr>
<tr>
<td>HRECO5</td>
<td>1859</td>
</tr>
<tr>
<td>HRECO6</td>
<td>1859</td>
</tr>
<tr>
<td>HRECO7</td>
<td>1859</td>
</tr>
<tr>
<td>IMPLEMENT</td>
<td>1855</td>
</tr>
<tr>
<td>INCSPR1</td>
<td>1431</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCH</td>
<td>5239#</td>
</tr>
<tr>
<td>INLOW</td>
<td>5241 5246#</td>
</tr>
<tr>
<td>INIT</td>
<td>1490#</td>
</tr>
<tr>
<td>INITLP</td>
<td>1410# 1423</td>
</tr>
<tr>
<td>INPIND</td>
<td>2167# 2190</td>
</tr>
<tr>
<td>INPDR</td>
<td>1035 2178# 2490</td>
</tr>
<tr>
<td>INPKY</td>
<td>1543 1675 1828 1846 2196 2234 2463 2588 2658# 3115 3119</td>
</tr>
<tr>
<td>INVALS</td>
<td>1346# 1381 1417</td>
</tr>
<tr>
<td>ITMP</td>
<td>706# 1557 1590 1595 1597 1625 1626 1646 1647 1705 1712 1715 1725 1732 1761 1830 1832 1837</td>
</tr>
<tr>
<td>JGORES</td>
<td>2408 2429#</td>
</tr>
<tr>
<td>JNFTBL</td>
<td>2385 2399#</td>
</tr>
<tr>
<td>JTOFIL</td>
<td>2402 2426#</td>
</tr>
<tr>
<td>JTORG</td>
<td>2481 2424#</td>
</tr>
<tr>
<td>JTOIST</td>
<td>2493 2419#</td>
</tr>
<tr>
<td>JTOVPR</td>
<td>2400 2419#</td>
</tr>
<tr>
<td>JTOVREG</td>
<td>2404 2412#</td>
</tr>
<tr>
<td>JTOUSER</td>
<td>2405 2418#</td>
</tr>
<tr>
<td>KDPDBUF</td>
<td>1212# 2334 2540 5629 5811 6161</td>
</tr>
<tr>
<td>KBDIN</td>
<td>5081# 5816</td>
</tr>
<tr>
<td>KBDIN</td>
<td>2630 5799#</td>
</tr>
<tr>
<td>KDDPOL</td>
<td>3047 3186 6150#</td>
</tr>
<tr>
<td>KCRIR</td>
<td>1519# 1980</td>
</tr>
<tr>
<td>KEY</td>
<td>760# 1544 1593 1740 1843 2107 2202 2205 2322 2346 2468 2532 2597 2592 2613 2627 2632</td>
</tr>
<tr>
<td>KEYCLR</td>
<td>1505# 2232 2620</td>
</tr>
<tr>
<td>KEYDM</td>
<td>1504# 1923 1926</td>
</tr>
<tr>
<td>KEYEND</td>
<td>1501# 1545 1844 2206 2264 2461 2618 3117</td>
</tr>
<tr>
<td>KEYFIL</td>
<td>1499# 1983</td>
</tr>
<tr>
<td>KEYFLG</td>
<td>949# 5533 5671 5692</td>
</tr>
<tr>
<td>KEYGO</td>
<td>1512# 1982</td>
</tr>
<tr>
<td>KEYLOC</td>
<td>1230# 5550 5644 5660 5666</td>
</tr>
<tr>
<td>KEYLSY</td>
<td>1518# 1984</td>
</tr>
<tr>
<td>KEYMOD</td>
<td>1513# 1981</td>
</tr>
<tr>
<td>KEYNI</td>
<td>1500# 2203 2623 2704 3121</td>
</tr>
<tr>
<td>KEYPAT</td>
<td>1503# 1929</td>
</tr>
<tr>
<td>KEYPK</td>
<td>1508# 1923 1926</td>
</tr>
<tr>
<td>KEYREI</td>
<td>1506# 1985</td>
</tr>
<tr>
<td>KEYREG</td>
<td>1509# 1922</td>
</tr>
<tr>
<td>KEYVIL</td>
<td>1502# 1986</td>
</tr>
<tr>
<td>KEYVIR</td>
<td>1587# 1929</td>
</tr>
<tr>
<td>KgORES</td>
<td>1511# 1989</td>
</tr>
<tr>
<td>KSETB</td>
<td>1514# 1987</td>
</tr>
<tr>
<td>LASTKY</td>
<td>904# 5555 5595 5562 5623 5670</td>
</tr>
<tr>
<td>LDATRA</td>
<td>751# 1856 2111 2211 2227 2242 2436 2562 2585 2607 2614 2632 2628 2635 2919 3088 3211 3321 3444 3346 3367 3368 3526 3590 3629 3641 3640 3668 3666 3715 3868 4154 4157 4160 4662 4669 4785 5028 5033 5001 5078 5186 5112 6297 6454</td>
</tr>
<tr>
<td>LDBYTE</td>
<td>3917 3876</td>
</tr>
<tr>
<td>LFEBSI</td>
<td>4837 4999#</td>
</tr>
<tr>
<td>LFEBSII</td>
<td>4780 4781 4820#</td>
</tr>
<tr>
<td>LFEBSIII</td>
<td>4777 4797 4813 4832#</td>
</tr>
<tr>
<td>LFEINT</td>
<td>4779 4870#</td>
</tr>
<tr>
<td>LFEPM</td>
<td>4776 4783#</td>
</tr>
<tr>
<td>LFER0</td>
<td>4051 4654#</td>
</tr>
<tr>
<td>LFEREG</td>
<td>4779 4936#</td>
</tr>
<tr>
<td>LFERET</td>
<td>4773 4778#</td>
</tr>
<tr>
<td>LFETCH</td>
<td>2561 3067 4784#</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSGL0</td>
<td>0000C</td>
</tr>
<tr>
<td>RDELAY</td>
<td>003F</td>
</tr>
<tr>
<td>RECON</td>
<td>02CC</td>
</tr>
<tr>
<td>RECTYP</td>
<td>0042</td>
</tr>
<tr>
<td>REGC</td>
<td>0044</td>
</tr>
<tr>
<td>REDRG</td>
<td>0065</td>
</tr>
<tr>
<td>REKOR</td>
<td>0198</td>
</tr>
<tr>
<td>KINT</td>
<td>0011</td>
</tr>
<tr>
<td>ROTCONT</td>
<td>0085</td>
</tr>
<tr>
<td>ROTPAT</td>
<td>0082</td>
</tr>
<tr>
<td>RSOURCE</td>
<td>0012</td>
</tr>
<tr>
<td>SCAN3</td>
<td>077C</td>
</tr>
<tr>
<td>SCAN5</td>
<td>0798</td>
</tr>
<tr>
<td>SCAN8</td>
<td>0790</td>
</tr>
<tr>
<td>SEGMAP</td>
<td>0046</td>
</tr>
<tr>
<td>SING</td>
<td>001A</td>
</tr>
<tr>
<td>SIZE</td>
<td>0000</td>
</tr>
<tr>
<td>SIZECH</td>
<td>0011</td>
</tr>
<tr>
<td>SWAIH</td>
<td>0031</td>
</tr>
<tr>
<td>SWAILO</td>
<td>0030</td>
</tr>
<tr>
<td>STRCOM</td>
<td>001D</td>
</tr>
<tr>
<td>STRLOC</td>
<td>002C</td>
</tr>
<tr>
<td>STRMEM</td>
<td>0026</td>
</tr>
<tr>
<td>STRTMP</td>
<td>0040</td>
</tr>
<tr>
<td>STRUTL</td>
<td>0019</td>
</tr>
<tr>
<td>STRVAR</td>
<td>0500</td>
</tr>
<tr>
<td>TCRLO</td>
<td>01D2</td>
</tr>
<tr>
<td>TIINT</td>
<td>0/4E</td>
</tr>
<tr>
<td>TIIKET</td>
<td>0748</td>
</tr>
<tr>
<td>TIPOF</td>
<td>079C</td>
</tr>
<tr>
<td>TIOUT</td>
<td>0040</td>
</tr>
<tr>
<td>TYPE</td>
<td>0037</td>
</tr>
<tr>
<td>UPDA01</td>
<td>017C</td>
</tr>
<tr>
<td>UPDADR</td>
<td>0178</td>
</tr>
<tr>
<td>VERSNO</td>
<td>0029</td>
</tr>
<tr>
<td>WERR</td>
<td>0016</td>
</tr>
<tr>
<td>WDISP</td>
<td>0606</td>
</tr>
<tr>
<td>WDISP1</td>
<td>06EE</td>
</tr>
<tr>
<td>XPCODE</td>
<td>0007</td>
</tr>
<tr>
<td>XP1EST</td>
<td>0301</td>
</tr>
</tbody>
</table>

Assembly Complete. No Errors.

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>Address</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>ADD</td>
<td>Add two 16-bit numbers</td>
</tr>
<tr>
<td>0001</td>
<td>SUB</td>
<td>Subtract two 16-bit numbers</td>
</tr>
<tr>
<td>0002</td>
<td>MOV</td>
<td>Move data to a register</td>
</tr>
<tr>
<td>0003</td>
<td>SHL</td>
<td>Shift left</td>
</tr>
<tr>
<td>0004</td>
<td>SHR</td>
<td>Shift right</td>
</tr>
<tr>
<td>0005</td>
<td>INC</td>
<td>Increment a register</td>
</tr>
<tr>
<td>0006</td>
<td>DEC</td>
<td>Decrement a register</td>
</tr>
<tr>
<td>0007</td>
<td>CMP</td>
<td>Compare two values</td>
</tr>
<tr>
<td>0008</td>
<td>BGT</td>
<td>Branch if greater than</td>
</tr>
<tr>
<td>0009</td>
<td>BLT</td>
<td>Branch if less than</td>
</tr>
<tr>
<td>000A</td>
<td>BEQ</td>
<td>Branch if equal</td>
</tr>
<tr>
<td>000B</td>
<td>BRN</td>
<td>Branch to a relative address</td>
</tr>
<tr>
<td>000C</td>
<td>INT</td>
<td>Interrupt request</td>
</tr>
<tr>
<td>000D</td>
<td>CLI</td>
<td>Clear interrupt enable flag</td>
</tr>
<tr>
<td>000E</td>
<td>STI</td>
<td>Set interrupt enable flag</td>
</tr>
<tr>
<td>000F</td>
<td>RST</td>
<td>Return from interrupt</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1975.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>AND</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>AUX</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>BX</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>CLC</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>CMN</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>CMP</td>
<td>0111</td>
<td>0111</td>
</tr>
<tr>
<td>DEC</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>DIF</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>DIV</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>DPH</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>DPL</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>EQU</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>ESC</td>
<td>1110</td>
<td>1110</td>
</tr>
<tr>
<td>FCH</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>INT</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>LDX</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>LDU</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>MOV</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>NOP</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>ORH</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>POP</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>PUSH</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>RCH</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>RES</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>SCH</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>SHH</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>STH</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>STA</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>STI</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>STR</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>SUB</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>XCH</td>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1976.
<table>
<thead>
<tr>
<th>mnemonic</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>094</td>
<td>4289</td>
</tr>
<tr>
<td>RM</td>
<td>183</td>
<td>863</td>
</tr>
<tr>
<td>R</td>
<td>90</td>
<td>385</td>
</tr>
<tr>
<td>R</td>
<td>12</td>
<td>4312</td>
</tr>
<tr>
<td>R1</td>
<td>182</td>
<td>849</td>
</tr>
<tr>
<td>R3</td>
<td>183</td>
<td>849</td>
</tr>
<tr>
<td>R5</td>
<td>184</td>
<td>849</td>
</tr>
</tbody>
</table>

**Notes:**
- All mnemonics copyrighted © Intel Corporation 1976.
- Page 1-170
APPENDIX C
COMMAND SUMMARY

The following is a summary of the commands implemented by the HSE-49 emulator monitor. Within each command group, tokens in each column indicate options the user has when invoking those commands.

Tokens in square brackets indicate dedicated keys on the keyboard (some keys having shared functions); angle brackets enclose hex digit strings used to specify an address or data parameter. Parameters in parentheses are optional, with the effects explained above. The notation used is as follows:

- <SMA> — Starting Memory Address for block command,
- <EMA> — Ending Memory Address for block command,
- <LOC> — LOCATION for individual accesses,
- <DATA> — DATA byte.

Asterisks (*) indicate the default condition for each command; thus that token is optional and serves to regularize the command syntax.

Program/data entry and verification commands:

- [EXAM] [PROG MEM] <LOC> [NEXT]
- [DATA MEM] [PREV]
- [HWRE REG]
- [PROG BRK]
- [DATA BRK]

Program/data initialization commands:

- [FILL] [PROG MEM] <SMA> [EMA] [DATA]
- [DATA MEM]
- [REGISTER]
- [HWRE REG]
- [PROG BRK]
- [DATA BRK]

Intellec® development system or TTY interface commands (for transferring HEX format files):

- [UPLOAD] [PROG MEM] <SMA> [EMA] [DATA]
- [DATA MEM]
- [REGISTER]
- [HWRE REG]
- [PROG BRK]
- [DATA BRK]

- [DOWNLOAD] [PROG MEM] [DATA MEM]
- [REGISTER]
- [HWRE REG]
- [PROG BRK]
- [DATA BRK]

Formatted data dump to TTY or CRT:

- [LIST] [PROG MEM] <SMA> [EMA]
- [DATA MEM]
- [REGISTER]
- [HWRE REG]
- [PROG BRK]
- [DATA BRK]

Program execution commands:

- [GO] [NO BREAK] <SMA> [EMA]
- [SING STP]
- [AUTO BRK]
- [AUTO STP]
- [GO/RST] [NO BREAK] [DATA MEM]
- [PROG BRK]
- [DATA BRK]

Breakpoint setting and clearing:

- [SET BRK] [PROG MEM] <LOC> [DATA MEM]
- [DATA MEM]
- [CLR BRK] [PROG MEM] <LOC> [DATA MEM]

APPENDIX D
ERROR MESSAGES

The following error message codes are used by the monitor software to report an operator or hardware error. Errors may be cleared by pressing [CLR/PREV] or [END/]. The format used for reporting errors is “Error—.n” where “n” is a hex digit.

Operator Errors
1. Illegal command initiator.
2. Illegal command modifier or parameter digit.
3. Illegal terminator for Examine command.
4. Illegal attempt to clear Error mode.
5-9. Not used.

Hardware Errors
A. ASCII error — non-hex digit encountered in data field of hex format record.
B. Breakpoint error. Break logic activated though breakpoints not enabled.
C. Hex format record checksum error. Note — the checksum will not be verified if the first character of the checksum field is a question mark (“?”) rather than a hexadecimal digit. This allows object files to be patched using the ISIS text editor without the necessity of manually recomputing the checksum value.
D. Not used.
E. Execution processor failed to respond to a command or parameter passed to it by the master processor. EP automatically reset. EP internal status may be lost. Program memory not affected.
F. Not used.
Using the 8049 as an 80 Column Printer Controller

John Kataubsy
Applications Engineering
USING THE 8049 AS AN 80 COLUMN
PRINTER CONTROLLER

I. INTRODUCTION

This Application Note details using INTEL's 8049 microcomputer as a dot matrix printer controller. Previous INTEL Application notes, (e.g. AP-27 and AP-54) described using intelligent processors and peripherals to control single printer mechanisms. This Application note expands upon the theme established in these prior notes and extends the concept to include a complete bi-directional 80 column printer using a single line buffer. For convenience this application note is divided into six sections:

1. INTRODUCTION
2. PRINT MECHANISM DESCRIPTION
3. INTERFACE CIRCUITY
4. SOFTWARE
5. CONCLUSION
6. APPENDIX

Over the last few years 80 column output devices have become somewhat of a de facto output standard for business and some data processing applications. It should be mentioned that by no means is the 80 column format a "new" standard. 80 column computer cards have been around for more than 20 years and perhaps the existence of these cards in the early days of computers is why the 80 column format is a standard today.

Many CRT terminals use the 80 by N format and to complement this a number of printers use this same format. One reason, aside from those historic in nature, for the 80 column standard is that 80 columns of 12 pitch text on standard typewritten 8.5 inch by 11 inch paper completely fills up an entire line and allows ample room for margins. So, the 80 column format is an aesthetically convenient format.

Printers are usually divided into either impact or non-impact and a character or line oriented device. Impact printers actually use some type of "striker" to place ink on the paper. More often than not the ink is contained on a ribbon which is placed between the striker and the paper. Non-impact printers use some means other than direct pressure to place the characters on the paper. This type of printer is very fast because there is very little mechanical motion associated with placing the characters on the paper. However, because the paper is required to be treated with a special substance, it is not as convenient as an impact printer.

Character printers are capable of printing one character at a time. (Any standard home typewriter is in effect a character printer.) Line printers must print an entire line at a time. Line printers are usually quite a bit faster than character printers, but they usually don't offer the print quality of character printers.

In recent years, the "computer boom" has caused the price of printers to tumble markedly. High volume production, competition, and the tremendous demand for reliable print mechanisms have all contributed to the decrease in price. Because of their simplicity, line printer mechanisms have decreased in price faster than other mechanisms. Therefore, when high quality print is not needed, a line printer is a very attractive choice.

This application note describes how to control an 80 column impact-line printer with an 8049/8039. The complete software listing is included in the appendix. The 8049 is the high-performance member of the MCS-48™ microcontroller family. The Processor has all of the features of the 8048 plus twice the amount of program and data memory and an 11MHz clock speed. For details about the 8049, please refer to the MCS-48 user's manual.

II. PRINT MECHANISM DESCRIPTION

The model 820 printer is available from C. ITOH ELECTRONICS (5301 BEETHOVEN STREET, LOS ANGELES, CA 90066). This inexpensive and simple printer is ideal for applications requiring 80 columns of dot matrix alpha-numeric information.

The model 820 printer is comprised of three basic sub-assemblies; the chassis or frame, the paper feed mechanism, and the print head. The diagram in Figure 2.1 gives the physical dimensions of the basic print mechanism. The basic chassis for the printer is constructed out of four sheet metal stampings. These stampings are screwed together to form a sturdy base on which all other components of the printer are mounted.

The paper feed mechanism consists of a toothed wheel, a solenoid, a tension spring, and a "catcher." When the solenoid is activated, the arm of the solenoid pulls against the spring and drags over the toothed wheel. When the solenoid is released, its arm is pulled by the spring, but this time the arm grabs a tooth on the wheel and pulls the wheel forward which advances the paper. A "catcher," which is merely a piece of plastic held against the toothed wheel, is added to assure that the paper is advanced only one "tooth" position each time the solenoid is activated.

The print head is comprised of seven solenoids which are mounted in a common housing. The solenoids are physically mounted in a circle, but their hammers are positioned linearly along the vertical axis. These seven vertically positioned hammers are the strikers that actually do the printing.
A motor, mounted toward the back of the print mechanism, drives a rubber toothed belt which turns a roller guide. A motor turns a guide that moves the print head from right to left and left to right. By properly timing the current flow through the solenoids while the print head is moving across the paper, characters can be formed. Figure 2.2 illustrates how the dot matrix printer "forms" its characters.

The timing pulses for the print head mechanism are generated by an opto-electronic sensor. This sensor, located on the left side plate of the printer, informs the print controller when to apply current to the print head mechanism. This "on-board timing wheel" assures that all characters will be properly spaced and that they will all be "in-line" in a vertical sense.

The print mechanism is also equipped with two additional sensors. These are the left home position sensor, located near the left front of the mechanism, and the right home position sensor, located near the right front of the print mechanism. These sensors simply tell the controller when the print head is in either the left or right home position. A complete timing chart for the printer is shown in Figure 2.3.

### III. INTERFACE CIRCUITRY

The manual supplied with the printer recommends some specific interface circuitry. For the most part the circuitry used in this Application Note followed these suggestions. The circuitry needed to drive the print head solenoid is shown in Figure 3.1. This same
and converted to TTL levels in order to interface to the controller. This conversion is accomplished with a simple voltage comparator. Figure 3.2 is a schematic of the sensor interface circuitry. Note that hysteresis is employed on the voltage comparators. This eliminates "false" sensing.

Motor control is accomplished by using a Monsanto MCS-6200 optically-coupled TRIAC. This part is ideal in this kind of application because it provides a simple means of controlling a line-operated motor without sacrificing the isolation needed for safe and reliable operation. Figure 3.3 is a schematic of the motor driving circuit.

As mentioned in Section 2, the printer provides some sensor interface signals that are derived via three optoelectronic sensors. These signals must be amplified

---

**Figure 3.1 Solenoid Drive Circuit (Eliminate R2 for Line Feed Solenoid)**

Circuit is used to drive the line feed solenoid except that the current limiting resistor R2 is eliminated. This resistor is not needed because the line feed solenoid is physically much larger than the print head solenoids and can tolerate much higher levels of current.

The print head drivers are connected to an 8212 latch. The latch is interfaced to the BUS PORT on the 8049 and is enabled whenever the WR pin and the BIT 4 of PORT 1 are coincidentally low. The line feed driver is connected to PORT 1 BIT 1 of the 8049.

Note that the driver is simply a Darlington transistor that is driven by an open collector TTL gate. Resistor R2 is the current limiting resistor and diode D, capacitor C, and resistor R3 are used to "dampen" the inductive spike that occurs when driving solenoid S. This circuit is repeated for each of the seven solenoids in the print head. It should be mentioned that, although the type of Darlington transistor needed to drive the print head is not critical, a collector current rating of at least 5 amps and a breakdown voltage (Veoc) of at least 100 volts is needed. Transistors that do not meet these requirements will be damaged by the inductive kickback of the solenoids.

As mentioned in Section 2, the printer provides some sensor interface signals that are derived via three optoelectronic sensors. These signals must be amplified

---

**Figure 3.2 Example of Sensor Circuit**

Motor control is accomplished by using a Monsanto MCS-6200 optically-coupled TRIAC. This part is ideal in this kind of application because it provides a simple means of controlling a line-operated motor without sacrificing the isolation needed for safe and reliable operation. Figure 3.3 is a schematic of the motor driving circuit.
To interface 8049 to the outside world one 8212 latch was used. This latch was connected to the BUS PORT and is enabled by an INS or MOVX instruction coincident with BIT 4 of PORT 1 being in a logical zero state. In this configuration, the 8212 was used to hold the data until read by the 8049. The connection of the 8212 to the 8049 is shown in Figure 3.4 and the parallel port timing diagram is shown in Figure 3.5. The 8212 parallel port was connected to the LINE PRINTER OUTPUT of an INTELLEC MICROCOMPUTER DEVELOPMENT SYSTEM.

![Figure 3.4 Connection of the 8212 Input Port to the 8049](image)

**Figure 3.4 Connection of the 8212 Input Port to the 8049**

**Figure 3.5 Parallel Port Timing**

### IV. SOFTWARE

As mentioned in Section 2, the bulk of the timing needed to control the printer is actually generated by the printer itself. Therefore, all the software must do is harness these timing signals and turn on and off the right solenoids at the right time.

To make things easy, the software needed to drive the printer is broken into four separate routines. These are:

1. **INITIALIZATION ROUTINE**
2. **INPUT ROUTINE**
3. **OUTPUT ROUTINE**
4. **LOOKUP ROUTINE**

The INITIALIZATION ROUTINE turns the motor on and checks the opto-electronic sensors. If a failure is found, the routine turns off the motor and loops on itself. This insures that the print mechanism is cycled properly before characters are accepted for printing.

This routine also initializes all of the variables used by the printer.

The INPUT ROUTINE reads the characters that are present in the 8212 input port and writes them into the 8049's buffer memory. The routine then checks the characters to see if a CARRIAGE RETURN (ASCII 0CH) has been transmitted. If a CR is detected, the input routine automatically inserts a LINE FEED as the next character. When the input routine detects a LINE FEED, it stops reading characters and sets the direction bits and the print bit in the status register. This action evokes the OUTPUT ROUTINE. A detailed flowchart of the INPUT ROUTINE is shown in Figure 4.1.

![Figure 4.1 Input Routine Flowchart](image)

**Figure 4.1 Input Routine Flowchart**
The OUTPUT ROUTINE initializes both the input and output buffer pointers and then reads the characters from the 8049's buffer memory. After a character is read the OUTPUT ROUTINE calls the LOOKUP ROUTINE which reads the proper bit pattern to form that character. This bit pattern is then used to strobe the solenoids. After each character is printed, the OUTPUT ROUTINE calls the INPUT ROUTINE and another character is placed into the buffer memory. This type of operation guarantees that the input buffer cannot "overrun" the output buffer. A flowchart of the OUTPUT ROUTINE is shown in Figure 4.2.

Initially the input buffer pointer is loaded with the address of the first location in the buffer memory. As characters are read, the input buffer pointer increments and fills the memory as shown in Figure 4.3(b) through 4.3(f). When a CARRIAGE RETURN-LINE FEED (CRLF) is encountered the input buffer pointer and the output buffer pointer are reset back to the first location. The OUTPUT ROUTINE then reads the character from the first location in the buffer memory, increments the output buffer pointer and calls the INPUT ROUTINE, which reads another character from the parallel input port.

The OUTPUT ROUTINE reads the entire buffer, inserting space codes (20H) after a CR is detected, and the input buffer pointer follows the output buffer pointer as they "increment" up to the memory. When the OUTPUT ROUTINE has printed the last character or space, the output buffer pointer and the input buffer pointer are set to point at the last location of the buffer memory. The OUTPUT ROUTINE then reads the character from the last location of the buffer memory and proceeds to "decrement" down the buffer memory. Space codes are inserted until a CR is found. Figure 4.3(1) to 4.3(0).

The input buffer pointer follows the output buffer pointer just as in the previous case. When the last, or in this case the first character is printed, the output buffer pointer and the input buffer pointer are set to point at the last location of the buffer memory. Now the pointers are "decrementing" down the buffer memory, but the printer is actually printing in a "normal" left to right fashion.

When the last character or space is printed, the output buffer and the input buffer pointer are set to the first location of the buffer memory and printing takes place in a reverse or right to left manner. After this line is printed, the print head and both buffer pointers are in the same position as they were initially. So, four lines must be printed before the buffer pointers and the print head complete a cycle. Each of these situations is handled separately by four different subroutines: CASE0, CASE1, CASE2, and CASE3.

IV-II. TIMING

All critical timing for the printer controller came from two basic sources; the timing sensors on the printer and the internal eight-bit timer of the 8049.

The internal timer of the 8049 was used to control the length of time the solenoids were fired (600 microseconds) and was also used as a "one-shot" to align the printer. This alignment is needed to make the "backward" printing line up vertically with the normal or forward printing. The "one-shot" is used to measure the time from the last column of the last character position until the right sensor flag is covered.
<table>
<thead>
<tr>
<th>Figure 4.3</th>
<th>I/O Buffer Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3A</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3B</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3C</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3D</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3E</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3F</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3G</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3H</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3I</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3J</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3K</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3L</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3M</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3N</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
<tr>
<td>4.3O</td>
<td>OUTPUT BUFFER</td>
</tr>
<tr>
<td></td>
<td>BUFFER MEMORY</td>
</tr>
<tr>
<td></td>
<td>INPUT BUFFER</td>
</tr>
</tbody>
</table>
When the print head reverses direction and the right sensor flag is uncovered, the timer is then used to determine where to start printing in the reverse direction.

The timer and the print wheel on the printer are used to determine when to place a character. The strobe from the print wheel informs the 8049 when to fire the solenoids and the timer allows the proper spacing between the characters.

V. CONCLUSION

Although the full speed of the 8049 was not used in this application, the high speed of the 8049 makes it possible to “fine-tune” any critical timing parameters. Additionally, the extra available CPU time could be used to add an interrupt driven keyboard and display, such as the ones discussed in AP-40, to the printer. This would allow the printer to function as a complete “terminal”.

Very little attempt was made to optimize the software, but still the entire program fits easily in 1.25K of memory; 750 bytes for printer control and 500 bytes for character lookup. Adding lower case to the printer would require an additional 500 bytes of lookup table. The remaining 250 bytes should be used to add “user” features such as tabs, double width printing, etc.

The high speed of the 8049 combined with its hardware and software architecture make it an ideal choice for controlling an 80 column, bi-directional line printer. The I/O structure of the 8049 minimizes the amount of external hardware needed to control the printer and the large amount of on-board program and data memory allow quite a sophisticated control program to be implemented.
APPENDIX A. SCHEMATIC DIAGRAM

+5 VOLTS

11.0 MHZ XTAL
VTAL 1
VTAL 2

15pF

RESET

2 μF

8049
8039

P11 28
P10 27
P20 21
P21 22
P23 23

TO
PSEN
WR
RD
DB1 12
DB2 13
DB3 14
DB4 15
DB5 16
DB6 17
DB7 18

31 39 33 34

RIGHT SENSOR
LEFT SENSOR
POSITION SENSOR

DO1 DO2 DO3 DO4 DO5 DO6 DO7 DO8

1 2 3 4 5 6 7 8

A7 1 21
A6 2 19
A5 3 17
A4 4 15
A3 5 10
A2 6 8
A1 7 6
A0 8 4

OE

2716

A0
A1
A2
A3
A4
A5
A6
A7

DO0 DO1 DO2 DO3 DO4 DO5 DO6 DO7

D0 D1 D2 D3 D4 D5 D6 D7

1 2 3 4 5 6 7 8

8212

DS2 2

BUSY
DATA1 DATA2 DATA3 DATA4 DATA5 DATA6 DATA7

1 2 3 4 5 6 7 8

DO9 DO10 DO11 DO12

D8 D9 D10 D11

1 2 3 4

8212

DS1 INT

DO13 DO14 DO15 DO16

D12 D13 D14 D15

1 2 3 4

THE 8212 AND THE 2716 WOULD NOT BE NEEDED IF AN 8049 WAS USED INSTEAD OF AN 8039

Solenoid Driver #1
Solenoid Driver #2
Solenoid Driver #3
Solenoid Driver #4
Solenoid Driver #5
Solenoid Driver #6
Solenoid Driver #7
APPENDIX B. MONITOR LISTING

APPENDIX B. MONITOR LISTING

LOC OBJ SEQ SOURCE STATEMENT

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

081B0811~08

BD B3

BOO~

BBB6

Bsa,

aBOF

aS26

~Eg

s

~

10

J.I

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

:******************************************************************************

: THIS PROGRAM IMPLEMENTS CONTROL OF THE CITON MODEL 82B

: PRINTER THE HARDWARE CONFIGURATION IS AS SUCH:

: 8212 INPUT PORT ON BUS = DATA INPUT

: 8212 OUTPUT PORT ON BUS = OUTPUT TO SOLENOID HAMMERS

: TI: INPUT = CHARACTER POSITIONING SENSOR ON PRINTER

: TB: INPUT = INTERRUPT FROM 8212 INPUT PORT

: PORT 1B = MOTOR ON, LOW = ON

: PORT 1I = LINE FEED STROBE, LOW = ON

: PORT 16 = LEFT MARGIN SENSOR, LOW WHEN COVERED, HIGH WHEN OPEN

: PORT 17 = RIGHT MARGIN SENSOR, LOW WHEN COVERED, HIGH WHEN OPEN

: PORT 12 = PIN 2 OF LM339, PRINT WHEEL SENSOR

: PORT 14 = PIN 13 OF LM339

: PORT 15 = PIN 14 OF LM339

:******************************************************************************

: SYSTEM EQUATES

:******************************************************************************

: INPUT LOCATION

: OUTPUT LOCATION

: PRINTING

: BUFFER LOCATION

: PRINT DIRECTION

: PRINT DIRECTION

: BUFFER LOAD DIRECTION

: THE LINE COUNTER

: MAX BUFFER LOCATION

: BOTTOM OF BUFFER

:******************************************************************************
LOC OBJ  SEQ  SOURCE STATEMENT

0099   44  ORG 0099
45
46  JUMP OVER THE INTERRUPT LOCATIONS
47
00BA  49  DIS 1 :DON'T USE INTERRUPTS
00B1  4A00  JMP BLIN :BEGIN THE PROGRAM
50
00B4  52  ORG BAH
53
54  :START THE PROGRAM
55
56  LOOP UNTIL THE BUFFER FILLS UP
57
00BA F0  58 PRINT: MOV A, STATUS :GET THE STATUS
00BB F211  59 JRI LPRTNT :IF PRINTING, CONTINUE
00BD  60 CALL LD0UF :READ INTO THE BUFFER
0001  B48B  61 JMP PRTNT :LOOP
62
63  :THIS ROUTINE PRINTS A LINE
64  IT FIRST SAVES THE STATUS
65  AND THEN DETERMINES WHICH DIRECTION TO PRINT
66  AND HOW TO MANIPULATE THE BUFFER
67
0011  68 LPRTNT: JMP SAVCHK :GO FIX UP THE STATUS
0013  F224  69 LPRTNT: JRP CASE2J :JUMP TO CASE 2 AND 3
0015  B417  70 JMP CASEB1 :JUMP TO CASE B AND 1
71
72  (CASEB1). LOADING THE BUFFER FROM FIRST TO MAX
73
8B17  B92B  74 CASEB1: MOV OUTBUF,#FIRST :SET UP OUTBUF
8019  B92B  75 MOV INBUF,#FIRST :SET UP INBUF
801B  FA  76 MOV A, SAVPNT :GET THE SAVED STATUS
801C  940C  77 CALL M0T0R :TURN ON THE MOTOR
801E  9252  78 JRA CASE1 :PRINT FORWARD
8020  9403  79 CALL PRTNTBK :GET READY TO PRINT BACKWARDS
8022  B431  80 JMP CASEB :PRINT BACKWARDS
81
82  (CASE23). LOADING BUFFER FROM MAX TO FIRST
83
8024  896F  84 CASE23: MOV OUTBUF,#MAX :SET UP OUTBUF
8026  B90F  85 MOV INBUF,#MAX :SET UP INBUF
8028  FA  86 MOV A, SAVPNT :GET THE PRINT STATUS
8029  940C  87 CALL M0T0R :TURN ON THE MOTOR
802B  9403  88 JRB CASE3 :PRINT LEFT TO RIGHT
802D  9403  89 CALL PRTNTBK :GET READY TO PRINT BACKWARDS
802F  B40D  90 JMP CASE2 :PRINT RIGHT TO LEFT
91
92 EJECT
LOC        OBJ        SEQ  SOURCE STATEMENT

0031 F1     93 CASEB:  MOV A.ROUTBUF     GET THE CHARACTER
0032 3491    94 CALL    FKPRNT     ADJUST FOR PRINTING
0034 B12B    95 MOV @OUTBUF.528H PUT A SPACE IN BUFFER RAN
0036 F242    96 JB7 FDC     FOUND A CR
0038 945E    97 CALL    INCTST     UPDATE OUTBUF
003A C6AE    98 JZ WATCHD    WAIT FOR END
003C BF28    99 MOV JUNKI.528H GET A SPACE TO PRINT
003E 9463    100 CALL    GTPRNT     GD PRINT A SPACE
0040 8431    101 JMP CASEB     LOOP
0042 8F28    102 FDC: MOV JUNKI.528H GD PRINT THE LAST SPACE
0044 9463    103 FDC1: CALL GTPRNT GD PRINT A CHARACTER
0046 945E    104 CALL    INCTST     CHECK OUT BUFFER
0048 C6AE    105 JZ WATCHD    WAIT FOR THE END
004A F1     106 MOV A.ROUTBUF GET THE CHARACTER
004C B12B    107 MOV @OUTBUF.528H PUT A SPACE THERE
004E 3491    108 CALL    FKPRNT     FIX THE CHARACTER UP
0050 AF     109 MOV JUNKI.A     SAVE IT
0058 8444    110 JMP FDC1     LOOP

111 ;
112 ;
113 CASE I, PRINTING LEFT TO RIGHT, LOADING BUFFER FROM
114  ;FIRST TO MAX
115 ;
0B52 F1     116 CASEI:  MOV A.ROUTBUF     GET THE CHARACTER
0B53 3491    117 CALL    FKPRNT     ADJUST FOR PRINTING
0B55 AF     118 MOV JUNKI.A     SAVE ACC
0B56 B12B    119 MOV @OUTBUF.528H PUT A SPACE IN THE BUFFER
0B58 F262    120 JB7 CRFOND    FOUND A CR?
0B5A 945E    121 CALL    GTPRNT     GD PRINT THE CHARACTER
0B5C 945E    122 CALL    INCTST     CHECK THE BUFFER
0B5E C675    123 JZ WATCHD    IS THE LAST CHARACTER BEING PRINTED?
0B60 B452    124 JMP CASEI     LOOP
0B62 B12B    125 CRFOND: MOV @OUTBUF.528H PUT A SPACE IN THE BUFFER MEMORY
0B64 BF28    126 MOV JUNKI.528H PUT A SPACE IN TEMP LOCATION
0B66 9463    127 CALL    GTPRNT     GD PRINT THE SPACE
0B68 945E    128 CALL    INCTST     CHECK THE BUFFER
0B6A C675    129 JZ WATCHD    LAST CHARACTER PRINTED?
0B6C F1     130 MOV A.ROUTBUF GET THE NEXT CHARACTER
0B6E 3491    131 CALL    FKPRNT     ADJUST IT
0B6F 8462    132 JMP CRFOND    LOOP
133 $EJECT
; THIS ROUTINE CALLS THE LINE FEED
130  JMP  PRNT  ; GO BACK TO THE PRINT ROUTINE

; THIS ROUTINE COMPLETES A LINE WHEN THE PRINT HEAD IS MOVING LEFT TO RIGHT

; CALL LINEFD  ; STROBE LINE FEED SOLENOID

; CALL LBDBUF  ; READ THE LAST CHARACTER

; CALL A.OUTBUF  ; JUMP IF CONTINUE IS SET

; ANL A.@BFH  ; RESET BIT ONE

; ANL STATUS,A  ; RESTORE STATUS

; CALL JB2  ; DOLP  ; DO A LINE FEED IF BIT IS SET

; CALL A.OUTBUF  ; FXPRNT  ; ADJUST FOR PRINTING

; CALL A.@OUTBUF  ; GET THE CHARACTER

; CALL FPRT  ; ADJUST FOR PRINTING

; CALL @OUTBUF.2BH  ; PUT A SPACE IN BUFFER RAM

; FB7 FDCR  ; FIND A CR YET

; DOLP  ; CHECK THE BUFFER

; JB7 EXTCHD  ; IF ZERO WAIT FOR SENSOR FLAG

; JB5 DOLP  ; GO A SPACE WHERE THE CHARACTER WAS

; JZ  ; LEAVE IF DONE

; JZ  ; SAVE IT

; JB2  ; DOLP  ; PUT A SPACE WHERE THE CHARACTER WAS
LDC OBJ  SEQ  SOURCE STATEMENT

184                          ; THIS ROUTINE WAITS FOR THE SENSOR FLAGS TO BE COVERED
185                          ; WHEN PRINTING RIGHT TO LEFT
186                          ;
187                          ;
BBBB 3488                     188 WATCHD: CALL LDBUF: GO READ THE LAST CHARACTER
BBBB B9                       189 JB6 WATCHD: LOOP IF SENSOR IS NOT COVERED
BBBB D2AE                     190 JB6 MOV A. STATUS: SET THE STATUS
BBBB FD                       191 MOV A. STATUS, A: SET IF CONTINUE IS SET
BBBB 528A                     192 JB2 DVR: SEE IF CONTINUE IS SET
BBBB 94DF                     193 CALL MOTORF: TURN THE MOTOR OFF
BBBB 53FD                     194 ANL A, $0F6H: RESET BIT 1
BBBB 53F8                     195 ORV: ANL A, $0F6H: RESET BIT 3
BBBB AD                       196 MOV STATUS, A: RESTORE STATUS
BBBB FA                       197 MOV A. SAVPNT: GET THE SAVED STATUS
BBBB E271                      198 JB5 DOLF: ADD A LINE FEED
BBBB 840A                      199 JMP PRNT: EXIT

BBB2 F1                       200 CASE 3: PRINTING LEFT TO RIGHT. LOADING BUFFER FROM
201                          ; FIRST TO LAST

202                          ;
BBBB 3491                     203 CASE 3: MOV A. ROUTBUF: GET A CHARACTER
BBBB 3491                     204 CALL PRNT: FIX FOR PRINTING
BBBB 3491                     205 MOV JUNK1, A: SAVE CHARACTER
BBBB B120                     206 MOV ROUTBUF, $2BH: PUT A SPACE IN THE BUFFER
BBBB F202                     207 JB7 CRFND: LEAVE IF A CR IS FOUND
BBBB 9463                     208 CALL GTPRNT: GO PRINT THE CHARACTER
BBBB 9472                     209 CALL DECTST: CHECK THE BUFFER
BBBB C675                      210 JJ: WATCH: LEAVE IF DONE
BBBB 84C2                      211 JMP CASE 3: LOOP
BBBB B120                     212 CRFND: MOV ROUTBUF, $2BH: PUT A SPACE IN THE BUFFER AND
BBBB F202                     213 MOV JUNK1, $2BH: GET A SPACE
BBBB 9463                     214 CALL GTPRNT: PRINT A SPACE
BBBB 9472                     215 CALL DECTST: CHECK THE BUFFER
BBBB C675                      216 JJ: WATCH: LEAVE IF DONE
BBBB F1                       217 MOV A. ROUTBUF: GET NEXT CHARACTER
BBBB 3491                      218 CALL PRNT: ADJUST IT
BBBB 84D2                      219 JMP CRFND: LOOP

221 CRFND: MOV ROUTBUF, $2BH: PUT A SPACE IN THE BUFFER AND
222 MOV JUNK1, $2BH: GET A SPACE
223 CALL GTPRNT: PRINT A SPACE
224 CALL DECTST: CHECK THE BUFFER
225 JJ: WATCH: LEAVE IF DONE
226 MOV A. ROUTBUF: GET NEXT CHARACTER
227 CALL PRNT: ADJUST IT
228 JMP CRFND: LOOP

229 #EJECT

1-186
LOC OBJ SEQ SOURCE STATEMENT

81B 89 222 ORG 10BH

81B 89 223

81B 89 224 LDBUF: IN A.PI ;READ PORT 1

81B B21C 225 JBS LMOde ;BIT 5 = 1 = LINE MODE

81B 1287 226 JBB ARND ;JUMP AROUND IF MOTOR IS ON

81B 89B1 227 ORL 1.B;iBH ;TURN THE MOTOR OFF

81B 92BF 228 ARND: JBS HOFF ;NO FORM FEED

81B FE 229 MOV A.LINcHT ;GET THE LINE COUNTER

81B A4388 230 ORL A.ROB ;SET MSB

81B AE 231 MOV LINcHT.A ;RESTORE THE LINE COUNTER

81B 23FF 232 MOV A.ROBFI ;SET ACC

81B B21A 233 HOFF: JBS 3110F ;JUMP IF NO LINE FEED

81B 947B 234 CALL LINEFD ;CD DO A LF OR FF

81B 89 235 BUTLOP: IN A.PI ;READ THE PORT

81B B21A 236 JBB WOLF ;WAIT FOR SWITCH TO BE RELEASED

81B 921A 237 JB4 WOLF ;WAIT FOR SWITCH TO BE RELEASED

81B 2413 238 JMP BUTLOP ;LOOP

81B A448 239 WOLF: JMP LD_BUF ;LOOP

248 ; FIRST SEE IF A CHARACTER IS PRESENT IN THE BUFFER

81B C 261F 241 LD_MOdE: JNT8 CHAR ;IF CHARACTER PRESENT, READ IT

81E 83 244 RET ;IF NOT, EXIT ROUTINE

245 ;

246 ; IF THERE IS A CHARACTER, READ IT

247 ;

81F FD 248 CHAR: MOV A.STATUS ;GET THE STATUS

81B 5249 249 JB2 ARNDJP ;IF CONTINUE IS SET, DON'T LOAD

81B 9249 250 JB4 ARNDJP ;IF LF IS SET, DON'T LOAD

81B 724A 251 JB LFCRECK ;WAS CR SET, SEE IF NEXT CHAR IS LF

81B 9406 252 CALL CTCAR ;GD READ A CHARACTER

81B 3461 253 GOOD: CALL FCHAR ;MAKE SURE IT IS OK

81B B 254 MOV BUF, A ;SAVE CHARACTER IN BUFFER MEMORY

81B FD 255 MOV A.STATUS ;GET THE STATUS

81B 7239 256 JB7 SUBI ;IF BIT 7 IS SET DECREMENT BUFFER

81B 18 257 INC INBUF ;UPDATE INBUF

81B 237B 258 JB8 MOV A.FIRST=1 ;GET TOP

81B 60 259 XRL A,INBUF ;ARE WE AT THE TOP?

81B 9649 260 JB2 ARNDJP ;IF NOT GET THE STATUS

81B 89 261 MOV A.INBUF ;GET INBUF

81B B21A 262 DEC A ;CHANGE BY ONE

81B A8 263 MOV INBUF, A ;PUT IT BACK

81B 2449 264 JMP ARNDJP ;GET THE STATUS

81B 83 265 SUBI: MOV A.INBUF ;GET INBUF

81B B21B 266 DEC A ;CHANGE BY ONE

81B 8A 267 MOV INBUF, A ;PUT INBUF BACK

81B 231F 268 MOV A.FIRST=1 ;GET THE BOTTOM OF THE BUFFER

81B 60 269 XRL A,INBUF ;TEST THE BUFFER

81B 9649 270 JB2 ARNDJP ;IF NOT ZERO READ THE STATUS

81B 724A 271 INC INBUF ;MOVE INBUF BACK

81B 2449 272 JMP ARNDJP ;GO GET STATUS

81B FD 273 GETST: MOV A.STATUS ;GET THE STATUS

81B 1429 274 JB8 ARNDJP ;IF BIT B SET, BYPASS

81B 925B 275 JB4 STBITI ;IF LF IS FOUND, SET THE STATUS

81B 93 276 ARNDJP: RET ;EXIT

277 ; THIS ROUTINE "FORCES" A LF AFTER A CR

279 ;

81A 9406 280 LFCRECK: CALL CTCAR ;READ A CHARACTER

81A C 23B8 281 MOV A.ROB ;GET A LINE FEED

81E 2428 282 JMP GOOD ;JUMP BACK

283 ;

284 ; THIS ROUTINE SETS THE STATUS BITS

285 ;

81B 8D 286 STBITI: MOV A.STATUS ;LOAD THE STATUS

81B 3259 287 JB1 STPRNT ;IF STILL PRINTING, LEAVE

81B 4382 288 ORL A.R82H ;SET PRINT BIT

81B 8348 289 ADD A.ROB ;UPDATE POSITION COUNTER

81B A418 290 MOV STATUS, A ;PUT STATUS BACK

81B 83 291 RET ;EXIT ROUTINE

81B 5268 292 STPRNT: JB2 BYEBYE ;CHECK continue BIT

81B 4384 293 ORL A.ROB ;SET CONTINUE BIT

81B 8348 294 ADD A.ROB ;UPDATE PRINT DIRECTION

81B 83 295 MOV STATUS, A ;PUT THE STATUS BACK

81B 83 296 BYEBYE: RET ;EXIT

297 ;

1-187
LOC OBJ SEQ SOURCE STATEMENT
298 1:THIS ROUTINE "CONVERTS" LOWER CASE LETTERS TO
299 1:UPPER CASE
300
301 8161 97 281 FCMAR 2 CLN C 1:CLEAR THE CARRY
302 8162 537F 02 AML A.07FH 1:STRIP MSB
303 8164 4F 08 MDV JUNK1.A 1:SAVE ACC
304 8165 B3AB 04 ADD A.0B8H 1:SEE IF NUMBER IS 68H
305 8167 E67B 05 JNC FINI 1:IF CARRY ISN'T SET, JUMP
306 8169 FF 07 MDV A.JUNK1 1:GET ACC BACK
307 816A 37 08 CPL A 1:SUBTRACT 28H FROM THE ACC
308 816B 8328 09 ADD A.028H
309 816D 37 0A CPL A
310 816E 247A 0B JMP FIXDUN 1:JUMP TO TEST CR LF
311 8170 37 0C FINE: CPL A 1:NOw SUBTRACT ABH FROM ACC
312 8171 B3AB 0D ADD A.0B8H
313 8173 37 0E CPL A
314 8174 4F 10 FIXDUN: MOV JUNK1.A 1:SAVE A
315 8175 D38D 11 XRL A.0D8H 1:IS CHARACTER A CR
316 8177 967F 12 JNZ LFTEST 1:IF IT IS NOT TEST LF
317 8179 FD 13 MDV A.STATUS 1:GET THE STATUS
318 817A 4388 14 ORL A.088H 1:SET BIT 3
319 817C AD 15 MOV STATUS.A 1:RESTORE THE STATUS
320 817E 24BF 16 JMP FIXFIN 1:LEAVE
321 817F FF 17 LFTEST: MOV A.JUNK1 1:GET CHARACTER BACK
322 8180 D38A 18 XRL A.088H 1:IS IT A LF
323 8182 C689 19 JZ FIXUP 1:IF ITS NOT, WE ARE DONE
324 8184 FF 1A MDV A.JUNK1 1:GET THE CHARACTER BACK
325 8185 D38C 1B XRL A.0BCH 1:IS IT A FORM FEED
326 8187 96BF 1C JNZ FIXFIN 1:IF NOT FORM FEED, JUMP
327 8189 FD 1D MOV A.STATUS 1:GET THE STATUS
328 818A 431B 1E ORL A.018H 1:SET BIT 4
329 818C AD 1F MOV STATUS.A 1:RETURN THE STATUS
330 818D 3458 20 CALL STBITI 1:SET THE STATUS
331 818F FF 21 MOV A.JUNK1 1:GET THE CHARACTER
LOC OBJ SEQ SOURCE STATEMENT
332 8198 63 22 RET 1:EXIT FCMAR
333
334 1:THIS ROUTINE RECOGNIZES A LF, FF, AND CR
335 1:DURING THE PRINT OPERATION
336 1:IT ALSO FORCES A SPACE IF A CHARACTER FOUND
337 1:IN THE BUFFER IS NOT IN THE LOOKUP TABLE
338
340 819A D38C 24 XRL A.0D8H 1:FORM FEED
341 819C 682 25 JZ FFFIX 1:GO SET FORM FEED
342 819E FF 26 MOV A.JUNK1 1:RESTORE CHARACTER
343 819F D38D 27 XRL A.0BDH 1:SEE IF IT IS A CR
344 81A0 C68B 28 JZ CRFIX 1:LEAVE IF IT IS
345 81A2 FF 29 MOV A.JUNK1 1:GET ACC BACK
346 81A3 D38A 30 XRL A.0BDH 1:SEE IF IT IS A LF
347 81A5 C68B 31 JZ LFFIX 1:LEAVE IF IT IS
348 81A6 FF 32 MOV A.JUNK1 1:GET CHARACTER BACK
349 81A8 53EB 33 ANL A.0E8H 1:SEE IF IT IS A CHARACTER
350 81A9 96BD 34 JNZ ISCCHAR 1:IF IT IS JUMP
351 81A5 232B 35 MOV A.028H 1:PUT A SPACE IN ACC
352 81A7 63 36 RET 1:EXIT
353 81A8 438B 37 CRFIX: ORL A.088H 1:SET BIT 7
354 81A9 63 38 RET 1:EXIT
355 81A8 FD 39 LFFIX: MOV A.STATUS 1:GET THE STATUS
356 81AC 432B 40 ORL A.028H 1:SET LF BIT IN STATUS
357 81AE AD 41 MOV STATUS.A 1:PUT THE STATUS BACK
358 81AF 232B 42 MOV A.028H 1:GET A SPACE
359 81B1 63 43 RET 1:EXIT
360 81B2 FD 44 FFFIX: MOV A.STATUS 1:GET THE STATUS
361 81B3 432B 45 ORL A.028H 1:SET LINE FEED BIT
362 81B4 FD 46 MOV STATUS.A 1:PUT THE STATUS BACK
363 81B6 FE 47 MOV A.LINCNT 1:GET THE LINE COUNT
364 81B7 438B 48 ORL A.088H 1:SET BIT 7
365 81B9 AE 49 MOV LINCNT.A 1:PUT LINE COUNT BACK
366 81BA 232B 50 MOV A.028H 1:GET A SPACE
367 81BC 63 51 RET 1:EXIT
368 81BD FF 52 ISCCHAR: MOV A.JUNK1 1:GET CHARACTER BACK
369 81BE 533F 53 ANL A.03FH 1:STRIP THE TWO MSB
36A 81BE 63 54 RET 1:EXIT

1-188
LOC OBJ  SEW  SOURCE STATEMENT

371 ;  THIS ROUTINE PRINTS THE CHARACTER IN THE ACC
372 ;
373 ;
B1C1 AC 374 PRINT: MOV  TEMP1,A  SAVE CHARACTER
B1C2 E7 375 RL  A  MULTIPLY BY TWO
B1C3 E7 376 RL  A  MULTIPLY BY FOUR
B1C4 6C 377 ADD  A,TEMP1  ADD ONCE TO MULTIPLY BY 5
378 ;
379 ; NOW SEE WHAT PART OF THE LOOKUP TABLE TO USE
380 ;
B1C5 2C 381 XCH  A,TEMP1  PUT CHARACTER IN A, TARGET IN TEMP1
B1C6 22CA 382 JSR  SHORT  JUMP TO HIGH ADDRESS IF BIT 5 SET
B1C8 44AB 383 JMP  PAGE1  GO TO FIRST PART OF LOOKUP TABLE
B1C9 64AB 384 SHORT: JMP  PAGE2  GO TO SECOND PAGE OF LOOKUP TABLE
385 ;
386 ; THIS ROUTINE TRIGGERS THE SOLENOIDS FOR 688 MICROSECONDS
387 ; AFTER WAITING FOR THE TRIGGER SIGNAL FROM THE PRINTER
388 ;
B1CC AF 389 FIRE: MOV  JUNK1,A  SAVE THE ACC
B1CD FO 390 MOV  A,STATUS  GET THE STATUS
B1CE 2D04 391 JB6  HT1  SEE IF FORWARD OR BACKWARDS
B1DB 56DB 392 FIREX: JT1  FIREX  WAIT FOR TI
B1D2 2406 393 JMP  FIREY  LEAVE
B1D4 46D4 394 HT1: JMTI  HTI  LOOP
B1D6 FF 395 FIREY: MOV  A,JUNK1  GET ACC BACK
B1D7 9B 396 MOVX  PRB,A  TRIGGER THE SOLENOID
397 ;
398 ; NOW KILL 688 MICROSECONDS
399 ;
B1DB 23F3 408 MOV  A,PRF3H  LOAD DELAY NUMBER
B1DB 62 40B MOV  T,A  PUT IT IN TIM
B1D9 55 412 STRT  T  START THE TIMER
B1DC 16EB 413 TSJTF: JTF  KTDUN  LOOP ON TIMER FLAG
B1DE 24DC 414 JMP  TSJTF  ;
B1E8 27 415 KTDUN: CLR  A  ZERO ACC
B1E1 9B 416 MOVX  PRB,A  TURN OFF SOLENOIDS
B1E2 63 417 STOP  TCNT  STOP THE TIMER
B1E3 83 41B RET  EXIT FIRE ROUTINE
409 REJECT

1-189
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>418</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>419</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>420</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>421</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>422</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>423</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>424</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>425</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>426</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>427</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>428</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>429</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>430</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>431</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>432</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>433</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>434</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>435</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>436</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>437</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>438</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>439</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>440</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>441</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>442</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>443</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>444</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>445</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>446</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>447</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>448</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>449</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>450</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>452</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>453</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>454</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>455</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>456</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B28H</th>
<th>ORG</th>
<th>200H</th>
</tr>
</thead>
<tbody>
<tr>
<td>421</td>
<td>TABLE1:</td>
<td>DB 3EH</td>
</tr>
<tr>
<td>422</td>
<td></td>
<td>DB 41H</td>
</tr>
<tr>
<td>423</td>
<td></td>
<td>DB 50H</td>
</tr>
<tr>
<td>424</td>
<td></td>
<td>DB 59H</td>
</tr>
<tr>
<td>425</td>
<td></td>
<td>DB 4EH</td>
</tr>
<tr>
<td>426</td>
<td></td>
<td></td>
</tr>
<tr>
<td>427</td>
<td></td>
<td>DB 7CH</td>
</tr>
<tr>
<td>428</td>
<td></td>
<td>DB 12H</td>
</tr>
<tr>
<td>429</td>
<td></td>
<td>DB 11H</td>
</tr>
<tr>
<td>430</td>
<td></td>
<td>DB 12H</td>
</tr>
<tr>
<td>431</td>
<td></td>
<td>DB 7CH</td>
</tr>
<tr>
<td>432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>433</td>
<td></td>
<td>DB 7FH</td>
</tr>
<tr>
<td>434</td>
<td></td>
<td>DB 49H</td>
</tr>
<tr>
<td>435</td>
<td></td>
<td>DB 49H</td>
</tr>
<tr>
<td>436</td>
<td></td>
<td>DB 49H</td>
</tr>
<tr>
<td>437</td>
<td></td>
<td>DB 36H</td>
</tr>
<tr>
<td>438</td>
<td></td>
<td></td>
</tr>
<tr>
<td>439</td>
<td></td>
<td>DB 3EH</td>
</tr>
<tr>
<td>440</td>
<td></td>
<td>DB 41H</td>
</tr>
<tr>
<td>441</td>
<td></td>
<td>DB 41H</td>
</tr>
<tr>
<td>442</td>
<td></td>
<td>DB 41H</td>
</tr>
<tr>
<td>443</td>
<td></td>
<td>DB 22H</td>
</tr>
<tr>
<td>444</td>
<td></td>
<td></td>
</tr>
<tr>
<td>445</td>
<td></td>
<td>DB 7FH</td>
</tr>
<tr>
<td>446</td>
<td></td>
<td>DB 41H</td>
</tr>
<tr>
<td>447</td>
<td></td>
<td>DB 41H</td>
</tr>
<tr>
<td>448</td>
<td></td>
<td>DB 41H</td>
</tr>
<tr>
<td>449</td>
<td></td>
<td>DB 3EH</td>
</tr>
<tr>
<td>450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>451</td>
<td></td>
<td>DB 7FH</td>
</tr>
<tr>
<td>452</td>
<td></td>
<td>DB 49H</td>
</tr>
<tr>
<td>453</td>
<td></td>
<td>DB 49H</td>
</tr>
<tr>
<td>454</td>
<td></td>
<td>DB 49H</td>
</tr>
<tr>
<td>455</td>
<td></td>
<td>DB 41H</td>
</tr>
</tbody>
</table>

**NOTE:**

This is the lookup table. The MSB is not used; the MSB is the dot that is the top of any given character and the LSB is the dot that is the bottom of any given character.

8288 419 828H ORC 288H

8211 3E 421 TABLE1: DB 3EH ; *****

8212 41 422 DB 41H ; * *

8213 5D 423 DB 50H ; * *** *

8214 59 424 DB 59H ; * *** *

8215 4E 425 DB 4EH ; * *** *

8216 7C 426 DB 7CH ; *****

8217 12 427 DB 12H ; * *

8218 11 428 DB 11H ; * *

8219 12 429 DB 12H ; * *

821A 7C 430 DB 7CH ; *****

821B 7F 431 DB 7FH ; *****

821C 49 432 DB 49H ; * *

821D 49 433 DB 49H ; * *

821E 49 434 DB 49H ; * *

821F 36 435 DB 36H ; ** **

8220 3E 436 DB 3EH ; *****

8221 41 437 DB 41H ; * *

8222 41 438 DB 41H ; * *

8223 22 439 DB 22H ; * *

8224 41 440 DB 41H ; * *

8225 41 441 DB 41H ; * *

8226 41 442 DB 41H ; * *

8227 22 443 DB 22H ; * *

8228 7F 444 DB 7FH ; *****

8229 41 445 DB 41H ; * *

822A 41 446 DB 41H ; * *

822B 41 447 DB 41H ; * *

822C 41 448 DB 41H ; * *

822D 41 449 DB 41H ; * *

822E 3E 450 DB 3EH ; *****

822F 7F 451 DB 7FH ; *****

8230 49 452 DB 49H ; * *

8231 49 453 DB 49H ; * *

8232 49 454 DB 49H ; * *

8233 41 455 DB 41H ; * *

8234 41 456 #EJECT
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B21E 7F</td>
<td>458</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B21F 99</td>
<td>459</td>
<td>DB</td>
<td>89H</td>
</tr>
<tr>
<td>B220 89</td>
<td>460</td>
<td>DB</td>
<td>89H</td>
</tr>
<tr>
<td>B221 89</td>
<td>461</td>
<td>DB</td>
<td>89H</td>
</tr>
<tr>
<td>B222 81</td>
<td>462</td>
<td>DB</td>
<td>81H</td>
</tr>
<tr>
<td>B223 3E</td>
<td>463</td>
<td>DB</td>
<td>3EH</td>
</tr>
<tr>
<td>B224 41</td>
<td>464</td>
<td>DB</td>
<td>41H</td>
</tr>
<tr>
<td>B225 41</td>
<td>465</td>
<td>DB</td>
<td>41H</td>
</tr>
<tr>
<td>B226 51</td>
<td>466</td>
<td>DB</td>
<td>51H</td>
</tr>
<tr>
<td>B227 71</td>
<td>467</td>
<td>DB</td>
<td>71H</td>
</tr>
<tr>
<td>B228 7F</td>
<td>468</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B229 BB</td>
<td>469</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B22A BB</td>
<td>470</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B22B 7F</td>
<td>471</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B22D BB</td>
<td>472</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B22E 41</td>
<td>473</td>
<td>DB</td>
<td>41H</td>
</tr>
<tr>
<td>B22F 7F</td>
<td>474</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B230 41</td>
<td>475</td>
<td>DB</td>
<td>41H</td>
</tr>
<tr>
<td>B231 BB</td>
<td>476</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B232 2B</td>
<td>477</td>
<td>DB</td>
<td>2BH</td>
</tr>
<tr>
<td>B233 4B</td>
<td>478</td>
<td>DB</td>
<td>4BH</td>
</tr>
<tr>
<td>B234 4B</td>
<td>479</td>
<td>DB</td>
<td>4BH</td>
</tr>
<tr>
<td>B235 4B</td>
<td>480</td>
<td>DB</td>
<td>4BH</td>
</tr>
<tr>
<td>B236 3F</td>
<td>481</td>
<td>DB</td>
<td>3FH</td>
</tr>
<tr>
<td>B237 7F</td>
<td>482</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B238 BB</td>
<td>483</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B239 14</td>
<td>484</td>
<td>DB</td>
<td>14H</td>
</tr>
<tr>
<td>B23A 22</td>
<td>485</td>
<td>DB</td>
<td>22H</td>
</tr>
<tr>
<td>B23B 41</td>
<td>486</td>
<td>DB</td>
<td>41H</td>
</tr>
<tr>
<td>B23C 7F</td>
<td>487</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B23D 4B</td>
<td>488</td>
<td>DB</td>
<td>4BH</td>
</tr>
<tr>
<td>B23E 4B</td>
<td>489</td>
<td>DB</td>
<td>4BH</td>
</tr>
<tr>
<td>B23F 4B</td>
<td>490</td>
<td>DB</td>
<td>4BH</td>
</tr>
<tr>
<td>B240 4B</td>
<td>491</td>
<td>DB</td>
<td>4BH</td>
</tr>
<tr>
<td>B241 7F</td>
<td>492</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B242 BB</td>
<td>493</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B243 BC</td>
<td>494</td>
<td>DB</td>
<td>8CH</td>
</tr>
<tr>
<td>B244 B2</td>
<td>495</td>
<td>DB</td>
<td>82H</td>
</tr>
<tr>
<td>B245 7F</td>
<td>496</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B246 7F</td>
<td>497</td>
<td>DB</td>
<td>7FH</td>
</tr>
<tr>
<td>B247 BB</td>
<td>498</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B248 BB</td>
<td>499</td>
<td>DB</td>
<td>88H</td>
</tr>
<tr>
<td>B249 18</td>
<td>500</td>
<td>DB</td>
<td>18H</td>
</tr>
<tr>
<td>B24A 7F</td>
<td>501</td>
<td>DB</td>
<td>7FH</td>
</tr>
</tbody>
</table>

511 #EJECT
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B249</td>
<td>3E</td>
<td>512</td>
<td>DB 3EH</td>
</tr>
<tr>
<td>B24C</td>
<td>41</td>
<td>513</td>
<td>DB 41H</td>
</tr>
<tr>
<td>B24D</td>
<td>41</td>
<td>514</td>
<td>DB 41H</td>
</tr>
<tr>
<td>B24E</td>
<td>41</td>
<td>515</td>
<td>DB 41H</td>
</tr>
<tr>
<td>B24F</td>
<td>3E</td>
<td>516</td>
<td>DB 3EH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>517</td>
<td></td>
</tr>
<tr>
<td>B258</td>
<td>7F</td>
<td>518</td>
<td></td>
</tr>
<tr>
<td>B259</td>
<td>89</td>
<td>519</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>B25A</td>
<td>19</td>
<td>520</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>521</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>522</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>523</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>524</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>525</td>
<td>DB 3EH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>526</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>527</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>528</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>529</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>530</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>531</td>
<td>DB 7FH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>532</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>533</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>534</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>535</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>536</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>537</td>
<td>DB 2EH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>538</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>539</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>540</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>541</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>542</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>543</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>544</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>545</td>
<td>DB 7FH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>546</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>547</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>548</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>549</td>
<td>DB 3FH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>550</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>551</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>552</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>553</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>554</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>555</td>
<td>DB 1FH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>556</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>557</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>558</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>559</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>560</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>561</td>
<td>DB 7FH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>562</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>563</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>564</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>565</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>566</td>
<td></td>
</tr>
</tbody>
</table>

566 EJECT
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>8278</td>
<td>63</td>
<td>567</td>
<td>DB 63H</td>
</tr>
<tr>
<td>8279</td>
<td>14</td>
<td>569</td>
<td>DB 14H</td>
</tr>
<tr>
<td>827a</td>
<td>88</td>
<td>570</td>
<td>DB 88H</td>
</tr>
<tr>
<td>827b</td>
<td>14</td>
<td>571</td>
<td>DB 14H</td>
</tr>
<tr>
<td>827c</td>
<td>63</td>
<td>572</td>
<td>DB 63H</td>
</tr>
<tr>
<td>827d</td>
<td>83</td>
<td>573</td>
<td>DB 83H</td>
</tr>
<tr>
<td>827e</td>
<td>84</td>
<td>574</td>
<td>DB 84H</td>
</tr>
<tr>
<td>827f</td>
<td>79</td>
<td>575</td>
<td>DB 79H</td>
</tr>
<tr>
<td>8280</td>
<td>84</td>
<td>576</td>
<td>DB 84H</td>
</tr>
<tr>
<td>8281</td>
<td>83</td>
<td>577</td>
<td>DB 83H</td>
</tr>
<tr>
<td>8282</td>
<td>61</td>
<td>578</td>
<td>DB 61H</td>
</tr>
<tr>
<td>8283</td>
<td>51</td>
<td>581</td>
<td>DB 51H</td>
</tr>
<tr>
<td>8284</td>
<td>49</td>
<td>582</td>
<td>DB 49H</td>
</tr>
<tr>
<td>8285</td>
<td>45</td>
<td>583</td>
<td>DB 45H</td>
</tr>
<tr>
<td>8286</td>
<td>43</td>
<td>584</td>
<td>DB 43H</td>
</tr>
<tr>
<td>8287</td>
<td>7F</td>
<td>585</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>8288</td>
<td>7F</td>
<td>586</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>8289</td>
<td>41</td>
<td>587</td>
<td>DB 41H</td>
</tr>
<tr>
<td>8290</td>
<td>41</td>
<td>588</td>
<td>DB 41H</td>
</tr>
<tr>
<td>8291</td>
<td>41</td>
<td>589</td>
<td>DB 41H</td>
</tr>
<tr>
<td>8292</td>
<td>41</td>
<td>590</td>
<td>DB 41H</td>
</tr>
<tr>
<td>8293</td>
<td>82</td>
<td>591</td>
<td>DB 82H</td>
</tr>
<tr>
<td>8294</td>
<td>84</td>
<td>592</td>
<td>DB 84H</td>
</tr>
<tr>
<td>8295</td>
<td>84</td>
<td>593</td>
<td>DB 84H</td>
</tr>
<tr>
<td>8296</td>
<td>10</td>
<td>594</td>
<td>DB 18H</td>
</tr>
<tr>
<td>8297</td>
<td>10</td>
<td>595</td>
<td>DB 18H</td>
</tr>
<tr>
<td>8298</td>
<td>20</td>
<td>596</td>
<td>DB 20H</td>
</tr>
<tr>
<td>8299</td>
<td>20</td>
<td>597</td>
<td>DB 20H</td>
</tr>
<tr>
<td>829a</td>
<td>41</td>
<td>598</td>
<td>DB 41H</td>
</tr>
<tr>
<td>829b</td>
<td>41</td>
<td>599</td>
<td>DB 41H</td>
</tr>
<tr>
<td>829c</td>
<td>41</td>
<td>600</td>
<td>DB 41H</td>
</tr>
<tr>
<td>829d</td>
<td>7F</td>
<td>601</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>829e</td>
<td>7F</td>
<td>602</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>829f</td>
<td>1B</td>
<td>603</td>
<td>DB 18H</td>
</tr>
<tr>
<td>829g</td>
<td>1B</td>
<td>604</td>
<td>DB 18H</td>
</tr>
<tr>
<td>829h</td>
<td>1B</td>
<td>605</td>
<td>DB 18H</td>
</tr>
<tr>
<td>829i</td>
<td>1B</td>
<td>606</td>
<td>DB 18H</td>
</tr>
<tr>
<td>829j</td>
<td>1B</td>
<td>607</td>
<td>DB 18H</td>
</tr>
<tr>
<td>829k</td>
<td>1B</td>
<td>608</td>
<td>DB 18H</td>
</tr>
<tr>
<td>829l</td>
<td>1B</td>
<td>609</td>
<td>DB 18H</td>
</tr>
<tr>
<td>829m</td>
<td>1B</td>
<td>610</td>
<td>DB 48H</td>
</tr>
<tr>
<td>829n</td>
<td>1B</td>
<td>611</td>
<td>DB 48H</td>
</tr>
<tr>
<td>829o</td>
<td>1B</td>
<td>612</td>
<td>DB 48H</td>
</tr>
<tr>
<td>829p</td>
<td>1B</td>
<td>613</td>
<td>DB 48H</td>
</tr>
<tr>
<td>829q</td>
<td>1B</td>
<td>614</td>
<td>DB 48H</td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ</td>
<td>SEQ</td>
<td>SOURCE STATEMENT</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>------------------</td>
</tr>
<tr>
<td>616</td>
<td>;</td>
<td>617</td>
<td>PAGE1: MOV STBCHT, 0BBH : ZERO STROBE COUNTER</td>
</tr>
<tr>
<td>618</td>
<td>MOV</td>
<td>619</td>
<td>A, 3AVPMT : GET DIRECTION</td>
</tr>
<tr>
<td>620</td>
<td>CPL</td>
<td>621</td>
<td>A : FLIP BITS</td>
</tr>
<tr>
<td>622</td>
<td>JB6</td>
<td>623</td>
<td>BAKWRD : IF BACKWARD JUMP OUT</td>
</tr>
<tr>
<td>624</td>
<td>MDV</td>
<td>625</td>
<td>A, TEMPI : GET THE TARGET</td>
</tr>
<tr>
<td>626</td>
<td>A3</td>
<td>627</td>
<td>MOV A, PO : GET THE DATA</td>
</tr>
<tr>
<td>628</td>
<td>CALL</td>
<td>629</td>
<td>FIRE : STROBE THE SOLENOIDS</td>
</tr>
<tr>
<td>630</td>
<td>INC</td>
<td>631</td>
<td>TEMPI : INCREMENT THE POINTER</td>
</tr>
<tr>
<td>632</td>
<td>INC</td>
<td>633</td>
<td>STBCHT : INCREMENT THE STROBE COUNTER</td>
</tr>
<tr>
<td>634</td>
<td>XRL</td>
<td>635</td>
<td>A, #05H : IS IT FIVE</td>
</tr>
<tr>
<td>636</td>
<td>JNZ</td>
<td>637</td>
<td>SETTL : REPEAT IF NOT FIVE</td>
</tr>
<tr>
<td>638</td>
<td>ADD</td>
<td>639</td>
<td>A, #04H : COMPENSATE FOR GOING BACKWARDS</td>
</tr>
<tr>
<td>640</td>
<td>MDV</td>
<td>641</td>
<td>TEMPL, A : SAVE IT</td>
</tr>
<tr>
<td>642</td>
<td>MOV</td>
<td>643</td>
<td>A, TEMPI : GET THE DATA</td>
</tr>
<tr>
<td>644</td>
<td>CMP</td>
<td>645</td>
<td>A, #0JH : GET THE DATA</td>
</tr>
<tr>
<td>646</td>
<td>CALL</td>
<td>647</td>
<td>FIRE : STROBE THE SOLENOIDS</td>
</tr>
<tr>
<td>648</td>
<td>MOV</td>
<td>649</td>
<td>TEMPL, A : GET TEMPI</td>
</tr>
<tr>
<td>650</td>
<td>DEC</td>
<td>651</td>
<td>A : DECREASE BY ONE</td>
</tr>
<tr>
<td>652</td>
<td>MOV</td>
<td>653</td>
<td>TEMPL, A : PUT IT BACK</td>
</tr>
<tr>
<td>654</td>
<td>INC</td>
<td>655</td>
<td>STBCHT : INCREMENT THE STROBE COUNTER</td>
</tr>
<tr>
<td>656</td>
<td>XRL</td>
<td>657</td>
<td>A, #05H : IS IT FIVE</td>
</tr>
<tr>
<td>658</td>
<td>JNZ</td>
<td>659</td>
<td>SETTL : REPEAT IF NOT FIVE</td>
</tr>
<tr>
<td>660</td>
<td>JMP</td>
<td>661</td>
<td>SETTIM : GO BACK. CHARACTER IS DONE</td>
</tr>
</tbody>
</table>

1-194
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>643</td>
<td></td>
<td>646</td>
<td>ORC 388H</td>
</tr>
<tr>
<td>644</td>
<td></td>
<td>647</td>
<td>;:*</td>
</tr>
<tr>
<td>645</td>
<td></td>
<td>648</td>
<td>;:*</td>
</tr>
<tr>
<td>649</td>
<td>DB</td>
<td>651</td>
<td>DB 88H</td>
</tr>
<tr>
<td>650</td>
<td>DB</td>
<td>652</td>
<td>DB 88H</td>
</tr>
<tr>
<td>653</td>
<td>DB</td>
<td>654</td>
<td>DB 88H</td>
</tr>
<tr>
<td>655</td>
<td>DB</td>
<td>656</td>
<td>DB 88H</td>
</tr>
<tr>
<td>657</td>
<td>DB</td>
<td>658</td>
<td>DB 88H</td>
</tr>
<tr>
<td>659</td>
<td>DB</td>
<td>660</td>
<td>DB 88H</td>
</tr>
<tr>
<td>661</td>
<td>DB</td>
<td>662</td>
<td>DB 88H</td>
</tr>
<tr>
<td>663</td>
<td>DB</td>
<td>664</td>
<td>DB 88H</td>
</tr>
<tr>
<td>665</td>
<td>DB</td>
<td>666</td>
<td>DB 88H</td>
</tr>
<tr>
<td>667</td>
<td>DB</td>
<td>668</td>
<td>14H</td>
</tr>
<tr>
<td>669</td>
<td>DB</td>
<td>670</td>
<td>14H</td>
</tr>
<tr>
<td>671</td>
<td>DB</td>
<td>672</td>
<td>14H</td>
</tr>
<tr>
<td>673</td>
<td>DB</td>
<td>674</td>
<td>14H</td>
</tr>
<tr>
<td>675</td>
<td>DB</td>
<td>676</td>
<td>14H</td>
</tr>
<tr>
<td>677</td>
<td>DB</td>
<td>678</td>
<td>14H</td>
</tr>
<tr>
<td>679</td>
<td>DB</td>
<td>680</td>
<td>14H</td>
</tr>
<tr>
<td>681</td>
<td>DB</td>
<td>682</td>
<td>14H</td>
</tr>
<tr>
<td>683</td>
<td>DB</td>
<td>684</td>
<td>14H</td>
</tr>
<tr>
<td>685</td>
<td>DB</td>
<td>686</td>
<td>49H</td>
</tr>
<tr>
<td>687</td>
<td>DB</td>
<td>688</td>
<td>56H</td>
</tr>
<tr>
<td>689</td>
<td>DB</td>
<td>690</td>
<td>5BH</td>
</tr>
</tbody>
</table>

69B *EJECT
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0324</td>
<td>08</td>
<td>692</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0324</td>
<td>08</td>
<td>693</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0325</td>
<td>07</td>
<td>694</td>
<td>DB 87H</td>
</tr>
<tr>
<td>0326</td>
<td>08</td>
<td>695</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0327</td>
<td>08</td>
<td>696</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0328</td>
<td>01</td>
<td>697</td>
<td>DB 1CH</td>
</tr>
<tr>
<td>0329</td>
<td>22</td>
<td>699</td>
<td>DB 22H</td>
</tr>
<tr>
<td>032A</td>
<td>41</td>
<td>700</td>
<td>DB 41H</td>
</tr>
<tr>
<td>032B</td>
<td>08</td>
<td>701</td>
<td>DB 88H</td>
</tr>
<tr>
<td>032C</td>
<td>08</td>
<td>702</td>
<td>DB 88H</td>
</tr>
<tr>
<td>032D</td>
<td>08</td>
<td>703</td>
<td>DB 88H</td>
</tr>
<tr>
<td>032E</td>
<td>08</td>
<td>704</td>
<td>DB 88H</td>
</tr>
<tr>
<td>032F</td>
<td>41</td>
<td>705</td>
<td>DB 41H</td>
</tr>
<tr>
<td>0330</td>
<td>22</td>
<td>706</td>
<td>DB 22H</td>
</tr>
<tr>
<td>0331</td>
<td>0C</td>
<td>707</td>
<td>DB 1CH</td>
</tr>
<tr>
<td>0332</td>
<td>22</td>
<td>708</td>
<td>DB 22H</td>
</tr>
<tr>
<td>0333</td>
<td>14</td>
<td>709</td>
<td>DB 14H</td>
</tr>
<tr>
<td>0334</td>
<td>7F</td>
<td>710</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>0335</td>
<td>14</td>
<td>711</td>
<td>DB 14H</td>
</tr>
<tr>
<td>0336</td>
<td>22</td>
<td>712</td>
<td>DB 22H</td>
</tr>
<tr>
<td>0337</td>
<td>08</td>
<td>713</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0338</td>
<td>08</td>
<td>714</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0339</td>
<td>7F</td>
<td>715</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>033A</td>
<td>08</td>
<td>716</td>
<td>DB 88H</td>
</tr>
<tr>
<td>033B</td>
<td>08</td>
<td>717</td>
<td>DB 88H</td>
</tr>
<tr>
<td>033C</td>
<td>08</td>
<td>718</td>
<td>DB 88H</td>
</tr>
<tr>
<td>033D</td>
<td>0B</td>
<td>719</td>
<td>DB 88H</td>
</tr>
<tr>
<td>033E</td>
<td>0B</td>
<td>720</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0341</td>
<td>0B</td>
<td>721</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0342</td>
<td>0B</td>
<td>722</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0343</td>
<td>0B</td>
<td>723</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0344</td>
<td>0B</td>
<td>724</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0345</td>
<td>0B</td>
<td>725</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0346</td>
<td>0B</td>
<td>726</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0347</td>
<td>0B</td>
<td>727</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0348</td>
<td>0B</td>
<td>728</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0349</td>
<td>0B</td>
<td>729</td>
<td>DB 88H</td>
</tr>
<tr>
<td>034A</td>
<td>0B</td>
<td>730</td>
<td>DB 88H</td>
</tr>
<tr>
<td>034B</td>
<td>28</td>
<td>731</td>
<td>DB 28H</td>
</tr>
<tr>
<td>034C</td>
<td>18</td>
<td>732</td>
<td>DB 18H</td>
</tr>
<tr>
<td>034D</td>
<td>0B</td>
<td>733</td>
<td>DB 88H</td>
</tr>
<tr>
<td>034E</td>
<td>0B</td>
<td>734</td>
<td>DB 88H</td>
</tr>
<tr>
<td>034F</td>
<td>0B</td>
<td>735</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0350</td>
<td>3E</td>
<td>736</td>
<td>DB 3EH</td>
</tr>
<tr>
<td>0351</td>
<td>51</td>
<td>737</td>
<td>DB 51H</td>
</tr>
<tr>
<td>0352</td>
<td>49</td>
<td>738</td>
<td>DB 49H</td>
</tr>
<tr>
<td>0353</td>
<td>45</td>
<td>739</td>
<td>DB 45H</td>
</tr>
<tr>
<td>0354</td>
<td>3E</td>
<td>740</td>
<td>DB 3EH</td>
</tr>
<tr>
<td>0355</td>
<td>0B</td>
<td>741</td>
<td>DB 88H</td>
</tr>
<tr>
<td>0356</td>
<td>42</td>
<td>742</td>
<td>DB 42H</td>
</tr>
<tr>
<td>0357</td>
<td>7F</td>
<td>743</td>
<td>DB 7FH</td>
</tr>
<tr>
<td>0358</td>
<td>4B</td>
<td>744</td>
<td>DB 4BH</td>
</tr>
<tr>
<td>0359</td>
<td>0B</td>
<td>745</td>
<td>DB 88H</td>
</tr>
<tr>
<td>035A</td>
<td>62</td>
<td>746</td>
<td>DB 62H</td>
</tr>
<tr>
<td>035B</td>
<td>51</td>
<td>747</td>
<td>DB 51H</td>
</tr>
<tr>
<td>035C</td>
<td>49</td>
<td>748</td>
<td>DB 49H</td>
</tr>
<tr>
<td>035D</td>
<td>49</td>
<td>749</td>
<td>DB 49H</td>
</tr>
<tr>
<td>035E</td>
<td>46</td>
<td>750</td>
<td>DB 46H</td>
</tr>
<tr>
<td>035F</td>
<td>21</td>
<td>751</td>
<td>DB 21H</td>
</tr>
<tr>
<td>0360</td>
<td>41</td>
<td>752</td>
<td>DB 41H</td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ</td>
<td>SEW</td>
<td>SOURCE STATEMENT</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>------------------</td>
</tr>
<tr>
<td>B361</td>
<td>49</td>
<td>766</td>
<td>DB 49H</td>
</tr>
<tr>
<td>B362</td>
<td>40</td>
<td>767</td>
<td>DB 40H</td>
</tr>
<tr>
<td>B363</td>
<td>33</td>
<td>768</td>
<td>DB 33H</td>
</tr>
<tr>
<td>B364</td>
<td>18</td>
<td>770</td>
<td>DB 18H</td>
</tr>
<tr>
<td>B365</td>
<td>14</td>
<td>771</td>
<td>DB 14H</td>
</tr>
<tr>
<td>B366</td>
<td>12</td>
<td>772</td>
<td>DB 12H</td>
</tr>
<tr>
<td>B367</td>
<td>1F</td>
<td>773</td>
<td>DB 1FH</td>
</tr>
<tr>
<td>B368</td>
<td>18</td>
<td>774</td>
<td>DB 18H</td>
</tr>
<tr>
<td>B369</td>
<td>27</td>
<td>775</td>
<td>DB 27H</td>
</tr>
<tr>
<td>B36A</td>
<td>45</td>
<td>776</td>
<td>DB 45H</td>
</tr>
<tr>
<td>B36B</td>
<td>45</td>
<td>777</td>
<td>DB 45H</td>
</tr>
<tr>
<td>B36C</td>
<td>45</td>
<td>778</td>
<td>DB 45H</td>
</tr>
<tr>
<td>B36D</td>
<td>39</td>
<td>779</td>
<td>DB 39H</td>
</tr>
<tr>
<td>B36E</td>
<td>3C</td>
<td>780</td>
<td>DB 3CH</td>
</tr>
<tr>
<td>B36F</td>
<td>4A</td>
<td>781</td>
<td>DB 4AH</td>
</tr>
<tr>
<td>B370</td>
<td>49</td>
<td>782</td>
<td>DB 49H</td>
</tr>
<tr>
<td>B371</td>
<td>49</td>
<td>783</td>
<td>DB 49H</td>
</tr>
<tr>
<td>B372</td>
<td>31</td>
<td>784</td>
<td>DB 31H</td>
</tr>
<tr>
<td>B373</td>
<td>81</td>
<td>785</td>
<td>DB 81H</td>
</tr>
<tr>
<td>B374</td>
<td>71</td>
<td>786</td>
<td>DB 71H</td>
</tr>
<tr>
<td>B375</td>
<td>89</td>
<td>787</td>
<td>DB 89H</td>
</tr>
<tr>
<td>B376</td>
<td>85</td>
<td>788</td>
<td>DB 85H</td>
</tr>
<tr>
<td>B377</td>
<td>83</td>
<td>789</td>
<td>DB 83H</td>
</tr>
<tr>
<td>B378</td>
<td>36</td>
<td>790</td>
<td>DB 36H</td>
</tr>
<tr>
<td>B379</td>
<td>49</td>
<td>791</td>
<td>DB 49H</td>
</tr>
<tr>
<td>B37A</td>
<td>49</td>
<td>792</td>
<td>DB 49H</td>
</tr>
<tr>
<td>B37B</td>
<td>49</td>
<td>793</td>
<td>DB 49H</td>
</tr>
<tr>
<td>B37C</td>
<td>36</td>
<td>794</td>
<td>DB 36H</td>
</tr>
</tbody>
</table>

**789 SELECT**
<table>
<thead>
<tr>
<th>LOC OBJ</th>
<th>SEQ</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B37D 46</td>
<td>001</td>
<td>DB 46H 1 **</td>
</tr>
<tr>
<td>B37E 49</td>
<td>002</td>
<td>DB 49H 1 1 *</td>
</tr>
<tr>
<td>B37F 49</td>
<td>003</td>
<td>DB 49H 1 1 *</td>
</tr>
<tr>
<td>B380 29</td>
<td>004</td>
<td>DB 29H 1 3 *</td>
</tr>
<tr>
<td>B381 1E</td>
<td>005</td>
<td>DB 1EH 1 5 **</td>
</tr>
<tr>
<td>B382 00</td>
<td>007</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B383 00</td>
<td>008</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B384 14</td>
<td>009</td>
<td>DB 14H 1 1</td>
</tr>
<tr>
<td>B385 00</td>
<td>010</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B386 00</td>
<td>011</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B387 00</td>
<td>012</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B388 48</td>
<td>013</td>
<td>DB 48H 1 1 **</td>
</tr>
<tr>
<td>B389 34</td>
<td>014</td>
<td>DB 34H 1 3 **</td>
</tr>
<tr>
<td>B38A 00</td>
<td>015</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B38B 00</td>
<td>016</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B38C 00</td>
<td>017</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B38D 14</td>
<td>018</td>
<td>DB 14H 1</td>
</tr>
<tr>
<td>B38E 22</td>
<td>019</td>
<td>DB 22H 1 1 **</td>
</tr>
<tr>
<td>B38F 41</td>
<td>020</td>
<td>DB 41H 1 1 **</td>
</tr>
<tr>
<td>B390 00</td>
<td>021</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B391 14</td>
<td>022</td>
<td>DB 14H 1 1 **</td>
</tr>
<tr>
<td>B392 14</td>
<td>023</td>
<td>DB 14H 1 1 **</td>
</tr>
<tr>
<td>B393 14</td>
<td>024</td>
<td>DB 14H 1 **</td>
</tr>
<tr>
<td>B394 14</td>
<td>025</td>
<td>DB 14H 1 **</td>
</tr>
<tr>
<td>B395 14</td>
<td>026</td>
<td>DB 14H 1 **</td>
</tr>
<tr>
<td>B396 00</td>
<td>027</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B397 41</td>
<td>028</td>
<td>DB 41H 1 1 **</td>
</tr>
<tr>
<td>B398 22</td>
<td>029</td>
<td>DB 22H 1 1 **</td>
</tr>
<tr>
<td>B399 14</td>
<td>030</td>
<td>DB 14H 1 **</td>
</tr>
<tr>
<td>B39A 00</td>
<td>031</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B39B 00</td>
<td>032</td>
<td>DB 88H 1</td>
</tr>
<tr>
<td>B39C 01</td>
<td>033</td>
<td>DB 81H 1</td>
</tr>
<tr>
<td>B39D 59</td>
<td>034</td>
<td>DB 59H 1 **</td>
</tr>
<tr>
<td>B39E 05</td>
<td>035</td>
<td>DB 85H 1 **</td>
</tr>
<tr>
<td>B39F 02</td>
<td>036</td>
<td>DB 82H 1 **</td>
</tr>
</tbody>
</table>

EJECT
LOC OBJ SEQ SOURCE STATEMENT
0348 0BBB 843 PAGE2: MOV STBH, #00H : ZER0 STROBE COUNTER
0342 FA 844 MOV A, $5A0H : GET DIRECTION
0343 37 845 CPL A : FLIP BITS
0344 0205 846 JBC 8XWRD : IF BACKWARD JUMP OUT
0346 FC 847 LKH1: MOV A, TEMP1 : GET THE TARGET
0347 B368 848 ADD A, #68H : ADJUST THE TARGET
0349 A3 849 MOVF A, #0A : GET THE DATA
034A 34C1 850 CALL FIRE : STROBE THE SOLENOIDS
034C 1C 851 INC TEMP1 : INCREMENT THE POINTER
034D 18 852 INC STBCNT : INCREMENT THE STROBE COUNTER
03AE 8D8 853 MOV A, $8D8 : GET THE STROBE COUNTER
03AF D385 854 XRL A, #05H : IS IT FIVE
03B1 96A6 855 JMP LKH1 : REPEAT IF NOT FIVE
03B3 04AE 856 JMP SETTIM : GO BACK
03B5 FC 857 BKWRD: MOV A, TEMP1 : GET THE TARGET
03B6 B364 858 ADD A, #64H : COMPENSATE FOR GOING BACKWARDS
03BB AC 859 MOV TEMP1, A : SAVE IT
03BA FC 860 LKH11: MOV A, TEMP1 : GET THE TARGET
03B9 A3 861 MOVF A, #0A : GET THE DATA
03BB 34C1 862 CALL FIRE : STROBE THE SOLENOIDS
03BC FC 863 MOV A, TEMP1 : GET TEMP1
03BD 87 864 DEC A : DECREASE BY ONE
03BF AC 865 MOV TEMP1, A : PUT IT BACK
03CB 1B 866 INC STBCNT : INCREMENT THE STROBE COUNTER
03C1 FB 867 MOV A, $STBCNT : GET THE STROBE COUNTER
03C2 D385 868 XRL A, #05H : IS IT FIVE
03C4 96B9 869 JMP LKH1 : REPEAT IF NOT FIVE
03C6 04AE 870 JMP SETTIM : GO BACK. CHARACTER IS DONE

1-199
LOC OBJ  SEQ  SOURCE STATEMENT

0483 872  ORC #DBBH
0483 873
0483 874
0483 875 BEGIN: CLR A :ZERO ACC
0484 876 MOV A, #0FFH :TURN OFF THE SOLENOIDS
0484 877 MOV #38H :SET UP THE PRINTER
0484 878 CALL VARSET :SET UP THE SOFTWARE
0484 879 JMP PINT :GO START
0488 880
0488 881 SETUP: MOV A, #0FFH :LOAD ACC WITH VALUE TO TURN ON MOTOR
0488 882 OUTL P1A :TURN ON MOTOR
0488 883
0488 884 MOV DELAY 2.2 SECONDS WHILE CHECKING RIGHT SENSOR
0488 885
0488 886 MOV TEMPI :DBBH :LOAD DELAY VALUE ONE
0488 887 SELFC: MOV JUNKI :DBBH :LOAD DELAY VALUE TWO
0488 888 SELFBE: MOV LINCENT :DBBH :LOAD DELAY VALUE THREE
0488 889 SELFA: IN A, PI :READ PORT ONE
0488 890 SELFJ: IN A, PI :MAKE THINGS RIGHT
0488 891 JBF: DONER :IS BIT 7 SET?
0488 892 DJNZ JUNKI :SMALL LOOP
0488 893 DJNZ JUNKI :BIGGER LOOP
0488 894 DJNZ TEMPI :BIGGEST LOOP
0488 895 JMP ERRDR :SOMETHING IS WRONG
0488 896
0488 897 :NOW MAKE SURE THE RIGHT SENSOR IS CLEARED
0488 898
0488 899 MOVJ: DONER: MOV JUNKI :DBBH :SET UP DELAY
0488 900 SELFC: MOV LINCENT :DBBH :SMALL MORE DELAY
0488 901 SELFA: IN A, PI: GET THE FLAG INFORMATION
0488 902 JBF: DONER: IS FLAG CLEARED?
0488 903 DJNZ JUNKI :IF NOT LOOP
0488 904 DJNZ JUNKI :LOOP SOME MORE
0488 905 JMP ERRDR :CLEFT IF FLAG IS NOT UNCOVERED
0488 906
0488 907 :NOW CHECK THE LEFT SENSOR IN THE SAME MANNER AS THE
0488 908 :RIGHT SENSOR. EXCEPT DELAY ONLY 2.5 SECONDS
0488 909
0488 910 MOVJ: DONER: MOV TEMPI :DBBH :LOAD DELAY 1
0488 911 SELFBE: MOV JUNKI :DBBH :LOAD DELAY 2
0488 912 SELFBB: MOV LINCENT :DBBH :LOAD DELAY 3
0488 913 SELFA: IN A, PI :READ THE PORT
0488 914 CPL A :CHANGE THINGS AROUND
0488 915 JBF: DONEL: IN X, IF BIT 6 IS M ZERO
0488 916 DJNZ JUNKI: SMALL LOOP
0488 917 DJNZ JUNKI: BIGGER LOOP
0488 918 DJNZ TEMPI: BIGGEST LOOP
0488 919 JMP ERRDR: SOMETHING IS WRONG
0488 920 SELBE: ORL P1, #B0H: TURN MOTOR OFF
0488 921 RET :GO BACK
0488 922
0488 923 :NOW SET UP THE VARIABLES
0488 924
0488 925 VARSET: MOV A, #BFEH :LOAD THE TIMER
0488 926 MOV A, #00H
0488 927 STRT T :START THE TIMER
0488 928 MOV INBUF :FIRST :LOAD INPUT BUFFER
0488 929 MOV LINCENT :DBBH :SET LINE COUNT
0488 930 MOV STATUS :B0H :SET FORWARD BIT
0488 931
0488 932 :NOW CLEAR THE RAM AREA BY WRITING SPACE CODES
0488 933
0488 934 MOV OUTBUF :FIRST :LOAD OUTBUF
0488 935 CLRMEM: MOV A, #0BH :PUT SPACE CODE IN ACC
0488 936 MOV INBUF :ACC :PUT SPACE CODE IN DATA MEMORY
0488 937 INC OUTBUF :UPDATE THE POINTER
0488 938 MOV .OUTBUF :MOVE THE POINTER INTO ACC
0488 939 XOR A, #A0H: -1 :SEE IF DONE
0488 940 JKZ CLRMEM :LOOP IF NOT CLEARED
0488 941
0488 942 :NOW CLEAR THE 0212
0488 943
0488 944 ANL P1, #BFFH :SET ENABLE BIT
0488 945 MOVX A, INBUF :CLEAR THE 0212 INPUT BUFFER
0488 946 ORL P1, #B0H :RESET ENABLE BIT
0488 947

1-200
LOC OBJ  SEQ  SOURCE STATEMENT
8459 03  948  :NOW EXIT YRSET
949  :
8459 93  950  :RET  :LEAVE INITIALIZATION
951  :
8459 95  952  :THIS ROUTINE TURNS THE MOTOR OFF AND LOOPS
953  :
845A 9FF  954  :ERROR: ORL P1. #BFFH  :TURN OFF MOTOR
845C 845E  955  :DEAD: JMP DEAD  :LOOP BECAUSE SOMETHING IS WRONG
956  :
845B 90  957  :THESE ARE ALL SUBROUTINES THAT ARE CALLED
958  :
845F 19  959  :INCTST: INC OUTBUF  :UPDATE THE POINTER
8465 237B  960  :GET THE VALUE FOR THE LAST CHARACTER
8461 09  961  :XRL A. OUTBUF  :DO THE TEST
8462 03  962  :RET  :EXIT
8463 09  963  :GTPRNT: IN A. P1  :READ PORT ONE
8464 37  964  :CPL A  :FLIP BITS
8465 0263  965  :JOG GTPRNT  :LOOP UNTIL SENSOR IS UNCOVERED
8467 1660  966  :TSTJTF: JTF PIT  :SEE IF TIMER FLAG IS SET
8469 8467  967  :JMP TSTJTF  :TEST FLAG
846E 45  968  :PIT: STOP TCTM  :STOP THE TIMER
846F FF  969  MOV A. JUNK:  :GET THE CHARACTER
8470 34C1  970  CALL PRNTIT  :PRINT THE CHARACTER
847F 341E  971  CALL LMMDT  :GET ANOTHER CHARACTER
8471 03  972  :RET  :EXIT
8472 F9  973  :DELCTST: MOV A. OUTBUF  :GET OUTBUF
8473 07  974  DEC A  :REDUCE BY ONE
8474 97  975  MOV OUTBUF. A  :PUT BACK IN OUTBUF
8475 D31F  976  XRL A. #FIRST-1  :SEE IF IT IS ALL THE WAY DOWN
8477 03  977  :RET  :EXIT
978  :
979  :THIS ROUTINE DOES A LINE FEED
980  :
8478 FE  981  :LINEFEED: MOV A. LIMCHT  :GET THE LINE COUNT
847F 290  982  JBT DUFF  :IF BIT 7 IS SET DO A FORMFEED
847F 94F0  983  LFDO: WNL P1. #BDFH  :TURN ON THE SOLENOID
847F BC40  984  MOV TEMPI #80H  :LOAD ONE DELAY
847F BF33  985  LFPLP1: MOV JUNKI #93H  :LOAD ANOTHER DELAY
847F EF91  986  DJNZ JUNKI LFLP2  :LOOP
847F 0C7F  987  DJNZ TEMPI: LFLP1  :LOOP SOME MORE
8480 9902  988  ORL P1. #BCH  :TURN OFF LF SOLENOID
8480 0E  989  INC LIMCHT  :UPDATE THE LINE COUNTER
8480 FE  990  MOV A. LIMCHT  :GET THE LINE COUNT
8489 D326  991  XRL A. #03H  :IS PAGE DONE
8486 966F  992  JNZ NOTDON  :SKIP OVER
848B BEEE  993  MOV LIMCHT #BBH  :ZERO LINE COUNTER
994  :
995  :NOW DELAY 90 MILLISECONDS
996  :
848F BC00  997  NOTDON: MOV TEMPI #83H  :LOAD DELAY VALUES
8491 BFF0  998  LODP1: MOV JUNKI #8FFH  :
8493 EF93  999  LODP2: DJNZ JUNKI LOOP2  :GENERATE DELAY
8495 E51A  1000  DJNZ TEMPI: LOOP1  :
8497 03  1001  :RET  :LINE FEED IS DONE
1002  :
1003  :THIS ROUTINE DOES A FORM FEED
1004  :
8499 09  1005  DOFF: IN A P1  :GET THE STATUS
8499 37  1006  CPL A  :FLIP ACC
849A 53C0  1007  WNL A #BCDH  :LEAVE ONLY TWO MSB'S
849C C698  1008  JZ DOFF  :IF A FLAG ISN'T COVERED: LOOP
849E 09B1  1009  ORL P1. #BBH  :TURN THE MOTOR OFF
84A8 9478  1010  CALL LFDO  :GO DO ONE LINE FEED
84A2 FE  1011  FFXX: MOV A. LIMCHT  :GET THE LINE COUNT
84A6 537F  1012  WNL A #FFH  :STRIP BIT SEVEN
84A5 0388  1013  XPL A #B0H  :IS IT DONE
84A7 C646  1014  JZ FFDONE  :LEAVE IF IT IS
84A9 9478  1015  CALL LFDO  :PROBE THE SOLENOIDS
84AB 0442  1016  MOV FFLY  :CHECK THE FORM FEED OUT
84A9 03  1017  FFDONE: PET  :EXIT FORM FEED
1018  :
1019  :
101A 23E8  1019  SETTIM: MOV A #BB0H  :GET DELAY VALUE
1020  1020  :PUT IN TIMER
1021  1021  STRT T  :START THE TIMER
1022  1022  RET  :EXIT
An Introduction to the Intel®
MCS-51™ Single-Chip
Microcomputer Family

John Wharton
Microcontroller Applications

Intel Corporation 1980
1. INTRODUCTION

In 1976 Intel introduced the MCS-48™ family, consisting of the 8048, 8748, and 8035 microcomputers. These parts marked the first time a complete microcomputer system, including an eight-bit CPU, 1024 8-bit words of ROM or EPROM program memory, 64 words of data memory, I/O ports and an eight-bit timer-counter could be integrated onto a single silicon chip. Depending only on the program memory contents, one chip could control a limitless variety of products, ranging from appliances or automobile engines to text or data processing equipment. Follow-on products stretched the MCS-48™ architecture in several directions: the 8049 and 8039 doubled the amount of on-chip memory and ran 83% faster; the 8021 reduced costs by executing a subset of the 8048 instructions with a somewhat slower clock; and the 8022 put a unique two-channel 8-bit analog-to-digital converter on the same NMOS chip as the computer, letting the chip interface directly with analog transducers.

Now three new high-performance single-chip microcomputers—the Intel® 8051, 8751, and 8031—extend the advantages of Integrated Electronics to whole new product areas. Thanks to Intel's new HMOS technology, the MCS-51™ family provides four times the program memory and twice the data memory as the 8048 on a single chip. New I/O and peripheral capabilities both increase the range of applicability and reduce total system cost. Depending on the use, processing throughput increases by two and one-half to ten times.

This Application Note is intended to introduce the reader to the MCS-51™ architecture and features. While it does not assume intimacy with the MCS-48™ product line on the part of the reader, he/she should be familiar with some microprocessor (preferably Intel's, of course) or have a background in computer programming and digital logic.

Family Overview

Pinout diagrams for the 8051, 8751, and 8031 are shown in Figure 1. The devices include the following features:

- Single-supply 5 volt operation using HMOS technology.
- 4096 bytes program memory on-chip (not on 8031).
- 128 bytes data memory on-chip.
- Four register banks.
- 128 User-defined software flags.
- 64 Kilobytes each program and external RAM addressability.
- One microsecond instruction cycle with 12 MHz crystal.
- 32 bidirectional I/O lines organized as four 8-bit ports (16 lines on 8031).
- Multiple mode, high-speed programmable Serial Port.
- Two multiple mode, 16-bit Timer/Counters.
- Two-level prioritized interrupt structure.
- Full depth stack for subroutine return linkage and data storage.
- Augmented MCS-48™ instruction set.
- Direct Byte and Bit addressability.
- Binary or Decimal arithmetic.
- Signed-overflow detection and parity computation.
- Hardware Multiple and Divide in 4 usec.
- Integrated Boolean Processor for control applications.
- Upwardly compatible with existing 8048 software.
All three devices come in a standard 40-pin Dual In-Line Package, with the same pin-out, the same timing, and the same electrical characteristics. The primary difference between the three is the on-chip program memory—different types are offered to satisfy differing user requirements.

The 8751 provides 4K bytes of ultraviolet-Erasable, Programmable Read Only Memory (EPROM) for program development, prototyping, and limited production runs. (By convention, 1K means $2^{10} = 1024$. 1k—with a lower case “k”—equals $10^3 = 1000$.) This part may be individually programmed for a specific application using Intel’s Universal PROM Programmer (UPP). If software bugs are detected or design specifications change the same part may be “erased” in a matter of minutes by exposure to ultraviolet light and reprogrammed with the modified code. This cycle may be repeated indefinitely during the design and development phase.

The final version of the software must be programmed into a large number of production parts. The 8051 has 4K bytes of ROM which are mask-programmed with the customer’s order when the chip is built. This part is considerably less expensive, but cannot be erased or altered after fabrication.

The 8031 does not have any program memory on-chip, but may be used with up to 64K bytes of external standard or multiplexed ROMs, PROMs, or EPROMs. The 8031 fits well in applications requiring significantly larger or smaller amounts of memory than the 4K bytes provided by its two siblings.

(The 8051 and 8751 automatically access external program memory for all addresses greater than the 4096 bytes on-chip. The External Access input is an override for all internal program memory—the 8051 and 8751 will each emulate an 8031 when pin 31 is low.)

Throughout this Note, “8051” is used as a generic term. Unless specifically stated otherwise, the point applies equally to all three components. Table 1 summarizes the quantitative differences between the members of the MCS-48™ and MCS-51™ families.

The remainder of this Note discusses the various MCS-51™ features and how they can be used. Software and/or hardware application examples illustrate many of the concepts. Several isolated tasks (rather than one complete system design example) are presented in the hope that some of them will apply to the reader’s experiences or needs.

A document this short cannot detail all of a computer system’s capabilities. By no means will all the 8051 instructions be demonstrated; the intent is to stress new or unique MCS-51™ operations and instructions generally used in conjunction with each other. For additional hardware information refer to the Intel MCS-51™ Family User’s Manual, publication number 121517. The assembly language and use of ASM51, the MCS-51™ assembler, are further described in the MCS-51™ Macro Assembler User’s Guide, publication number 9800937.

The next section reviews some of the basic concepts of microcomputer design and use. Readers familiar with the 8048 may wish to skim through this section or skip directly to the next, “ARCHITECTURE AND ORGANIZATION.”

### Microcomputer Background Concepts

Most digital computers use the binary (base 2) number system internally. All variables, constants, alphanumeric characters, program statements, etc., are represented by groups of binary digits (“bits”), each of which has the value 0 or 1. Computers are classified by how many bits they can move or process at a time.

The MCS-51™ microcomputers contain an eight-bit central processing unit (CPU). Most operations process variables eight bits wide. All internal RAM and ROM, and virtually all other registers are also eight bits wide. An eight-bit (“byte”) variable (shown in Figure 2) may assume one of $2^8 = 256$ distinct values, which usually represent integers between 0 and 255. Other types of numbers, instructions, and so forth are represented by one or more bytes using certain conventions.

For example, to represent positive and negative values, the most significant bit (D7) indicates the sign of the other seven bits—0 if positive, 1 if negative—allowing integer variables between $-128$ and $+127$. For integers with extremely large magnitudes, several bytes are manipulated together as “multiple precision” signed or unsigned integers—16, 24, or more bits wide.

### Table 1. Features of Intel’s Single-Chip Microcomputers

<table>
<thead>
<tr>
<th>EPROM Program Memory</th>
<th>ROM Program Memory</th>
<th>External Program Memory</th>
<th>Program Memory (Int/Max)</th>
<th>Data Memory (Bytes)</th>
<th>Instr. Cycle Time</th>
<th>Input/Output Pins</th>
<th>Interrupt Sources</th>
<th>Reg. Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>8021</td>
<td>8022</td>
<td>8022</td>
<td>1K, 1K</td>
<td>64</td>
<td>8.4 μSec</td>
<td>21</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8748</td>
<td>8048</td>
<td>8035</td>
<td>2K, 2K</td>
<td>64</td>
<td>8.4 μSec</td>
<td>28</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8751</td>
<td>8049</td>
<td>8039</td>
<td>2K, 4K</td>
<td>128</td>
<td>2.5 μSec</td>
<td>27</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8051</td>
<td>8051</td>
<td>8031</td>
<td>4K, 64K</td>
<td>128</td>
<td>1.36 μSec</td>
<td>32</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

AFN-01502A-05

2-3
The letters “MCS” have traditionally indicated a system or family of compatible Intel® microcomputer components, including CPUs, memories, clock generators, I/O expanders, and so forth. The numerical suffix indicates the microprocessor or microcomputer which serves as the cornerstone of the family. Microcomputers in the MCS-48™ family currently include the 8048-series (8035, 8040, & 8748), the 8049-series (8039 & 8049), and the 8021 and 8022; the family also includes the 8243, an I/O expander compatible with each of the microcomputers. Each computer’s CPU is derived from the 8048, with essentially the same architecture, addressing modes, and instruction set, and a single assembler (ASM48) serves each.

The first members of the MCS-51™ family are the 8051, 8751, and 8031. The architecture of the 8051-series, while derived from the 8048, is not strictly compatible; there are more addressing modes, more instructions, larger address spaces, and a few other hardware differences. In this Application Note the letters “MCS-51” are used when referring to architectural features of the 8051-series—features which would be included on possible future microcomputers based on the 8051 CPU. Such products could have different amounts of memory (as in the 8048/8049) or different peripheral functions (as in the 8021 and 8022) while leaving the CPU and instruction set intact. ASM51 is the assembler used by all microcomputers in the 8051 family.

Two digit decimal numbers may be “packed” in an eight-bit value, using four bits for the binary code of each digit. This is called Binary-Coded Decimal (BCD) representation, and is often used internally in programs which interact heavily with human beings.

Alphanumeric characters (letters, numbers, punctuation marks, etc.) are often represented using the American Standard Code for Information Interchange (ASCII) convention. Each character is associated with a unique seven-bit binary number. Thus one byte may represent a single character, and a word or sequence of letters may be represented by a series (or “string”) of bytes. Since the ASCII code only uses 128 characters, the most significant bit of the byte is not needed to distinguish between characters. Often D7 is set to 0 for all characters. In some coding schemes, D7 is used to indicate the “parity” of the other seven bits—set or cleared as necessary to ensure that the total number of “1” bits in the eight-bit code is even (“even parity”) or odd (“odd parity”). The 8051 includes hardware to compute parity when it is needed.

A computer program consists of an ordered sequence of specific, simple steps to be executed by the CPU one-at-a-time. The method or sequence of steps used collectively to solve the user’s application is called an “algorithm.”

The program is stored inside the computer as a sequence of binary numbers, where each number corresponds to one of the basic operations (“opcodes”) which the CPU is capable of executing. In the 8051, each program memory location is one byte. A complete instruction consists of a sequence of one or more bytes, where the first defines the operation to be executed and additional bytes (if needed) hold additional information, such as data values or variable addresses. No instruction is longer than three bytes.

The way in which binary opcodes and modifier bytes are assigned to the CPU’s operations is called the computer’s “machine language.” Writing a program directly in machine language is time-consuming and tedious. Human beings think in words and concepts rather than encoded numbers, so each CPU operation and resource is given a name and standard abbreviation (“mnemonic”). Programs are more easily discussed using these standard mnemonics, or “assembly language,” and may be typed into an Intel® 8080 or Series 11® microcomputer development system in this form. The development system can mechanically translate the program from assembly language “source” form to machine language “object” code using a program called an “assembler.” The MCS-51™ assembler is called ASM51.

There are several important differences between a computer’s machine language and the assembly language used as a tool to represent it. The machine language or instruction set is the set of operations which the CPU can perform while a program is executing (“at run-time”), and is strictly determined by the microcomputer hardware design.

The assembly language is a standard (though more-or-less arbitrary) set of symbols including the instruction set mnemonics, but with additional features which further simplify the program design process. For example, ASM51 has controls for creating and formatting a program listing, and a number of directives for allocating variable storage and inserting arbitrary bytes of data into the object code for creating tables of constants.

### Figure 2. Representation of bits Within an Eight-Bit “Byte” (Value shown = 01010001 Binary = 81 decimal).
In addition, ASM51 can perform sophisticated mathematical operations, computing addresses or evaluating arithmetic expressions to relieve the programmer from this drudgery. However, these calculations can only use information known at "assembly time."

For example, the 8051 performs arithmetic calculations at run-time, eight bits at a time. ASM51 can do similar operations 16 bits at a time. The 8051 can only do one simple step per instruction, while ASM51 can perform complex calculations in each line of source code. However, the operations performed by the assembler may only use parameter values fixed at assembly-time, not variables whose values are unknown until program execution begins.

For example, when the assembly language source line,

```
ADD A,#(LOOP_COUNT + 1) * 3
```

is assembled, ASM51 will find the value of the previously-defined constant "LOOP_COUNT" in an internal symbol table, increment the value, multiply the sum by three, and (assuming it is between -256 and 255 inclusive) truncate the product to eight bits. When this instruction is executed, the 8051 ALU will just add that resulting constant to the accumulator.

Some similar differences exist to distinguish number system ("radix") specifications. The 8051 does all computations in binary (though there are provisions for then converting the result to decimal form). In the course of writing a program, though, it may be more convenient to specify constants using some other radix, such as base 10. On other occasions, it is desirable to specify the ASCII code for some character or string of characters without referring to tables. ASM51 allows several representations for constants, which are converted to binary as each instruction is assembled.

For example, binary numbers are represented in the assembly language by a series of ones and zeros (naturally), followed by the letter "B" (for Binary); octal numbers as a series of octal digits (0-7) followed by the letter "O" (for Octal) or "Q" (which doesn't stand for anything, but looks sort of like an "O" and is less likely to be confused with a zero).

Hexadecimal numbers are represented by a series of hexadecimal digits (0-9,A-F), followed by (you guessed it) the letter "H." A "hex" number must begin with a decimal digit; otherwise it would look like a user-defined symbol (to be discussed later). A "dummy" leading zero may be inserted before the first digit to meet this constraint. The character string "BACH" could be a legal label for a Baroque music synthesis routine; the string "0BACH" is the hexadecimal constant BAC. This is a case where adding 0 makes a big difference.

Decimal numbers are represented by a sequence of decimal digits, optionally followed by a "D." If a number has no suffix, it is assumed to be decimal—so it had better not contain any non-decimal digits. "0BAC" is not a legal representation for anything.

When an ASCII code is needed in a program, enclose the desired character between two apostrophes (as in "W") and the assembler will convert it to the appropriate code (in this case 23H). A string of characters between apostrophes is translated into a series of constants; 'BACH' becomes 42H, 41H, 43H, 48H. These same conventions are used throughout the associated Intel documentation. Table 2 illustrates some of the different number formats.

### 2. ARCHITECTURE AND ORGANIZATION

Figure 3 blocks out the MCS-51 internal organization. Each microcomputer combines a Central Processing Unit, two kinds of memory (data RAM plus program ROM or EPROM), Input/Output ports, and the mode.

#### Table 2. Notations Used to Represent Numbers

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Signed Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>0B</td>
<td>0Q</td>
<td>00H</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000000001</td>
<td>1B</td>
<td>1Q</td>
<td>01H</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>0000001111</td>
<td>111B</td>
<td>7Q</td>
<td>07H</td>
<td>7</td>
<td>+7</td>
</tr>
<tr>
<td>0000100000</td>
<td>1000B</td>
<td>10Q</td>
<td>08H</td>
<td>8</td>
<td>+8</td>
</tr>
<tr>
<td>0000100011</td>
<td>1001B</td>
<td>11Q</td>
<td>09H</td>
<td>9</td>
<td>+9</td>
</tr>
<tr>
<td>0000101000</td>
<td>1010B</td>
<td>12Q</td>
<td>0AH</td>
<td>10</td>
<td>+10</td>
</tr>
<tr>
<td>0000111111</td>
<td>1111B</td>
<td>17Q</td>
<td>0FH</td>
<td>15</td>
<td>+15</td>
</tr>
<tr>
<td>0001000000</td>
<td>1000B</td>
<td>20Q</td>
<td>10H</td>
<td>16</td>
<td>+16</td>
</tr>
<tr>
<td>0111111111</td>
<td>111111B</td>
<td>177Q</td>
<td>7FH</td>
<td>127</td>
<td>+127</td>
</tr>
<tr>
<td>1000000000</td>
<td>10000000B</td>
<td>200Q</td>
<td>80H</td>
<td>128</td>
<td>-128</td>
</tr>
<tr>
<td>1000000100</td>
<td>1000001B</td>
<td>201Q</td>
<td>81H</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>1111111100</td>
<td>1111110B</td>
<td>376Q</td>
<td>0FEH</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>1111111111</td>
<td>1111111B</td>
<td>377Q</td>
<td>0FFH</td>
<td>255</td>
<td>0</td>
</tr>
</tbody>
</table>
status, and data registers and random logic needed for a variety of peripheral functions. These elements communicate through an eight-bit data bus which runs throughout the chip, somewhat akin to indoor plumbing. This bus is buffered to the outside world through an I/O port when memory or I/O expansion is desired.

Let's summarize what each block does; later chapters dig into the CPU's instruction set and the peripheral registers in much greater detail.

Central Processing Unit
The CPU is the "brains" of the microcomputer, reading the user's program and executing the instructions stored therein. Its primary elements are an eight-bit Arithmetic/Logic Unit with associated registers A, B, PSW, and SP, and the sixteen-bit Program Counter and "Data Pointer" registers.
Arithmetic Logic Unit

The ALU can perform (as the name implies) arithmetic and logic functions on eight-bit variables. The former include basic addition, subtraction, multiplication, and division; the latter include the logical operations AND, OR, and Exclusive-OR, as well as rotate, clear, complement, and so forth. The ALU also makes conditional branching decisions, and provides data paths and temporary registers used for data transfers within the system. Other instructions are built up from these primitive functions: the addition capability can increment registers or automatically compute program destination addresses; subtraction is also used in decrementing or comparing the magnitude of two variables.

These primitive operations are automatically cascaded and combined with dedicated logic to build complex instructions such as incrementing a sixteen-bit register pair. To execute one form of the compare instruction, for example, the 8051 increments the program counter three times, reads three bytes of program memory, compares a register address with logical operations, reads internal data memory twice, makes an arithmetic comparison of two variables, computes a sixteen-bit destination address, and decides whether or not to make a branch—all in two microseconds!

An important and unique feature of the MCS-51 architecture is that the ALU can also manipulate one-bit as well as eight-bit data types. Individual bits may be set, cleared, or complemented, moved, tested, and used in logic computations. While support for a more primitive data type may initially seem a step backwards in an era of increasing word length, it makes the 8051 especially well suited for controller-type applications. Such algorithms inherently involve Boolean (true/false) input and output variables, which were heretofore difficult to implement with standard microprocessors. These features are collectively referred to as the MCS-51* "Boolean Processor," and are described in the so-named chapter to come.

Thanks to this powerful ALU, the 8051 instruction set fares well at both real-time control and data intensive algorithms. A total of 51 separate operations move and manipulate three data types: Boolean (1-bit), byte (8-bit), and address (16-bit). All told, there are eleven addressing modes—seven for data, four for program sequence control (though only eight are used by more than just a few specialized instructions). Most operations allow several addressing modes, bringing the total number of instructions (operation/addressing mode combinations) to 111, encompassing 255 of the 256 possible eight-bit instruction opcodes.

Instruction Set Overview

Table 4 lists these 111 instructions classified into five groups:

- Arithmetic Operations
- Logical Operations for Byte Variables
- Data Transfer Instructions
- Boolean Variable Manipulation
- Program Branching and Machine Control

MCS-48* programmers perusing Table 4 will notice the absence of special categories for Input/Output, Timer/Counter, or Control instructions. These functions are all still provided (and indeed many new functions are added), but as special cases of more generalized operations in other categories. To explicitly list all the useful instructions involving I/O and peripheral registers would require a table approximately four times as long.

Observant readers will also notice that all of the 8048's page-oriented instructions (conditional jumps, JMPP, MOV, MOV3) have been replaced with corresponding but non-paged instructions. The 8051 instruction set is entirely non-page-oriented. The MCS-48* "MOVP" instruction replacement and all conditional jump instructions operate relative to the program counter, with the actual jump address computed by the CPU during instruction execution. The "MOV3" and "JMPP" replacements are now made relative to another sixteen-bit register, which allows the effective destination to be anywhere in the program memory space, regardless of where the instruction itself is located. There are even three-byte jump and call instructions allowing the destination to be anywhere in the 64K program address space.

The instruction set is designed to make programs efficient both in terms of code size and execution speed. No instruction requires more than three bytes of program memory, with the majority requiring only one or two bytes. Virtually all instructions execute in either one or two instruction cycles—one or two microseconds with a 12-MHz crystal—with the sole exceptions (multiply and divide) completing in four cycles.

Many instructions such as arithmetic and logical functions or program control, provide both a short and a long form for the same operation, allowing the programmer to optimize the code produced for a specific application. The 8051 usually fetches two instruction bytes per instruction cycle, so using a shorter form can lead to faster execution as well.

For example, any byte of RAM may be loaded with a constant with a three-byte, two-cycle instruction, but the commonly used "working registers" in RAM may be initialized in one cycle with a two-byte form. Any bit anywhere on the chip may be set, cleared, or complemented by a single three-byte logical instruction using two cycles. But critical control bits, I/O pins, and software flags may be controlled by two-byte, single cycle instructions. While three-byte jumps and calls can "go anywhere" in program memory, nearby sections of code may be reached by shorter relative or absolute versions.
A significant side benefit of an instruction set more powerful than those of previous single-chip microcomputers is that it is easier to generate applications-oriented software. Generalized addressing modes for byte and bit operations reduce the number of source code lines written and debugged for a given application. This leads in turn to proportionately lower software costs, greater reliability, and faster design cycles.

**Accumulator and PSW**

The 8051, like its 8048 predecessor, is primarily an accumulator-based architecture: an eight-bit register called the accumulator ("A") holds a source operand and receives the result of the arithmetic instructions (addition, subtraction, multiplication, and division). The accumulator can be the source or destination for logical operations and a number of special data movement instructions, including table look-ups and external RAM expansion. Several functions apply exclusively to the accumulator: rotates, parity computation, testing for zero, and so on.

Many instructions implicitly or explicitly affect (or are affected by) several status flags, which are grouped together to form the Program Status Word shown in Figure 4.

(The period within entries under the Position column is called the “dot operator,” and indicates a particular bit position within an eight-bit byte. “PSW.5” specifies bit 5 of the PSW. Both the documentation and ASM51 use this notation.)

The most “active” status bit is called the carry flag (abbreviated “C”). This bit makes possible multiple precision arithmetic operations including addition, subtraction, and rotates. The carry also serves as a “Boolean accumulator” for one-bit logical operations and bit manipulation instructions. The overflow flag (OV) detects when arithmetic overflow occurs on signed integer operands, making two's complement arithmetic possible. The parity flag (P) is updated after every instruction cycle with the even-parity of the accumulator contents.

The CPU does not control the two register-bank select bits, RS1 and RS0. Rather, they are manipulated by software to enable one of the four register banks. The usage of the PSW flags is demonstrated in the Instruction Set chapter of this Note.

Even though the architecture is accumulator-based, provisions have been made to bypass the accumulator in common instruction situations. Data may be moved from any location on-chip to any register, address, or indirect address (and vice versa), any register may be loaded with a constant, etc., all without affecting the accumulator. Logical operations may be performed against registers or variables to alter fields of bits—without using or affecting the accumulator. Variables may be incremented, decremented, or tested without using the accumulator. Flags and control bits may be manipulated and tested without affecting anything else.

**Other CPU Registers**

A special eight-bit register (“B”) serves in the execution of the multiply and divide instructions. This register is used in conjunction with the accumulator as the second input operand and to return eight-bits of the result.

The MCS-51 family processors include a hardware stack within internal RAM, useful for subroutine linkage.
passing parameters between routines, temporary variable storage, or saving status during interrupt service routines. The Stack Pointer (SP) is an eight-bit pointer register which indicates the address of the last byte pushed onto the stack. The stack pointer is automatically incremented or decremented on all push or pop instructions and all subroutine calls and returns. In theory, the stack in the 8051 may be up to a full 128 bytes deep. (In practice, even simple programs would use a handful of RAM locations for pointers, variables, and so forth—reducing the stack depth by that number.) The stack pointer defaults to 7 on reset, so that the stack will start growing up from location 8, just like in the 8048. By altering the pointer contents the stack may be relocated anywhere within internal RAM.

Finally, a 16-bit register called the data pointer (DPTR) serves as a base register in indirect jumps, table look-up instructions, and external data transfers. The high- and low-order halves of the data pointer may be manipulated as separate registers (DPH and DPL, respectively) or together using special instructions to load or increment all sixteen bits. Unlike the 8048, look-up tables can therefore start anywhere in program memory and be of arbitrary length.

Memory Spaces
Program memory is separate and distinct from data memory. Each memory type has a different addressing mechanism, different control signals, and a different function.

The program memory array (ROM or EPROM), like an elephant, is extremely large and never forgets information, even when power is removed. Program memory is used for information needed each time power is applied: initialization values, calibration constants, keyboard layout tables, etc., as well as the program itself. The program memory has a sixteen-bit address bus; its elements are addressed using the Program Counter or instructions which generate a sixteen-bit address.

To stretch our analogy just a bit, data memory is like a mouse: it is smaller and therefore quicker than program memory, and it goes into a random state when electrical power is applied. On-chip data RAM is used for variables which are determined or may change while the program is running.

A computer spends most of its time manipulating variables, not constants, and a relatively small number of variables at that. Since eight-bits is more than sufficient to uniquely address 128 RAM locations, the on-chip RAM address register is only one byte wide. In contrast to the program memory, data memory accesses need a single eight-bit value—a constant or another variable—to specify a unique location. Since this is the basic width of the ALU and the different memory types, those resources can be used by the addressing mechanisms, contributing greatly to the computer's operating efficiency.

The partitioning of program and data memory is extended to off-chip memory expansion. Each may be added independently, and each uses the same address and data busses, but with different control signals. External program memory is gated onto the external data bus by the PSEN (Program Store Enable) control output, pin 29. External data memory is read onto the bus by the RD output, pin 17, and written with data supplied from the microcomputer by the WR output, pin 16. (There is no control pin to write external program ROM, which is by definition Read Only.) While both types may be expanded to up to 64K bytes, the external data memory may optionally be expanded in 256 byte "pages" to preserve the use of P2 as an I/O port. This is useful with a relatively small expansion RAM (such as the Intel® 8155) or for addressing external peripherals.

Single-chip controller programs are finalized during the project design cycle, and are not modified after production. Intel's single-chip microcomputers are not "von Neumann" architectures common among main-frame and mini-computer systems: the MCS-51® processor data memory—on-chip and external—may not be used for program code. Just as there is no write-control signal for program memory, there is no way for the CPU to execute instructions out of RAM. In return, this concession allows an architecture optimized for efficient controller applications: a large, fixed program located in ROM, a hundred or so variables in RAM, and different methods for efficiently addressing each.

(Von Neumann machines are helpful for software development and debug. An 8051 system could be modified to have a single off-chip memory space by gating together the two memory-read controls (PSEN and RD) with a two-input AND gate (Figure 5). The CPU could then write data into the common memory array using WR and
external data transfer instructions, and read instructions or data with the AND gate output and data transfer or program memory look-up instructions.)

In addition to the memory arrays, there is (yet) another (albeit sparsely populated) physical address space. Connected to the internal data bus are a score of special-purpose eight-bit registers scattered throughout the chip. Some of these—B, SP, PSW, DPH, and DPL—have been discussed above. Others—I/O ports and peripheral function registers—will be introduced in the following sections. Collectively, these registers are designated as the “special-function register” address space. Even the accumulator is assigned a spot in the special-function register address space for additional flexibility and uniformity.

Thus, the MCS-51™ architecture supports several distinct “physical” address spaces, functionally separated at the hardware level by different addressing mechanisms, read and write control signals, or both:

- On-chip program memory;
- On-chip data memory;
- Off-chip program memory;
- Off-chip data memory;
- On-chip special-function registers.

What the programmer sees, though, are “logical” address spaces. For example, as far as the programmer is concerned, there is only one type of program memory, 64K bytes in length. The fact that it is formed by combining on- and off-chip arrays (split 4K/60K on the 8051 and 8751) is “invisible” to the programmer; the CPU automatically fetches each byte from the appropriate array, based on its address.

(Presumably, future microcomputers based on the MCS-51™ architecture may have a different physical split, with more or less of the 64K total implemented on-chip. Using the MCS-48™ family as a precedent, the 8048’s 4K potential program address space was split 1K/3K between on- and off-chip arrays; the 8049’s was split 2K/2K.)

Why go into such tedious details about address spaces? The logical addressing modes are described in the Instruction Set chapter in terms of physical address spaces. Understanding their differences now will pay off in understanding and using the chips later.
Special Peripheral Functions

There are a few special needs common among control-oriented computer systems:

- keeping track of elapsed real-time;
- maintaining a count of signal transitions;
- measuring the precise width of input pulses;
- communicating with other systems or people;
- closely monitoring asynchronous external events.

Until now, microprocessor systems needed peripheral chips such as timer/counters, USARTs, or interrupt controllers to meet these needs. The 8051 integrates all of these capabilities on-chip!

Timer/Counters

There are two sixteen-bit multiple-mode Timer/Counters on the 8051, each consisting of a "High" byte (corresponding to the 8048 "T" register) and a low byte (similar to the 8048 prescaler, with the additional flexibility of being software-accessible). These registers are called, naturally enough, TH0, TH1, TL0, and TL1. Each pair may be independently software programmed to any of a dozen modes with a mode register designated TMOD (Figure 7), and controlled with register TCON (Figure 8).

The timer modes can be used to measure time intervals, determine pulse widths, or initiate events, with one-microsecond resolution, up to a maximum interval of 65,536 instruction cycles (over 65 milliseconds). Longer delays may easily be accumulated through software. Configured as a counter, the same hardware will accumulate external events at frequencies from D.C. to 500 KHz, with up to sixteen bits of precision.

Serial Port Interface

Each microcomputer contains a high-speed, full-duplex, serial port which is software programmable to function in four basic modes: shift-register I/O expander, 8-bit UART, 9-bit UART, or interprocessor communications link. The UART modes will interface with standard I/O devices (e.g. CRTs, teletypewriters, or modems) at data rates from 122 baud to 31 kilobaud. Replacing the standard 12 MHz crystal with a 10.7 MHz crystal allows 110 baud. Even or odd parity (if desired) can be included with simple bit-handling software routines. Inter-processor communications in distributed systems takes place at 187 kilobaud with hardware for automatic address/data message recognition. Simple TTL or CMOS shift registers provide low-cost I/O expansion at a super-fast 1 Mega-baud. The serial port operating modes are controlled by the contents of register SCON (Figure 9).

Interrupt Capability and Control

(Interrupt capability is generally considered a CPU function. It is being introduced here since, from an applications point of view, interrupts relate more closely to peripheral and system interfacing.)
GATE Gate control. When set, Timer/counter
"x" is enabled only while "INTx" pin is
high and "TRx" control bit is set. When
cleared, timer/counter is enabled
whenever "TRx" control bit is set.

C/T Timer or Counter Selector. Cleared for
Timer operation (input from internal
system clock). Set for Counter opera-
tion (input from "Tx" input pin).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF1</td>
<td>TCON.7</td>
<td>Timer 1 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.</td>
</tr>
<tr>
<td>TR1</td>
<td>TCON.6</td>
<td>Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off.</td>
</tr>
<tr>
<td>TF0</td>
<td>TCON.5</td>
<td>Timer 0 overflow Flag. Set by hardware on timer/counter overflow. Cleared when interrupt processed.</td>
</tr>
<tr>
<td>TR0</td>
<td>TCON.4</td>
<td>Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE1</td>
<td>TCON.3</td>
<td>Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.</td>
</tr>
<tr>
<td>IT1</td>
<td>TCON.2</td>
<td>Interrupt 1 Type control bit. Set cleared by software to specify falling edge low level triggered external interrupts.</td>
</tr>
<tr>
<td>IE0</td>
<td>TCON.1</td>
<td>Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.</td>
</tr>
<tr>
<td>IT0</td>
<td>TCON.0</td>
<td>Interrupt 0 Type control bit. Set cleared by software to specify falling edge low level triggered external interrupts.</td>
</tr>
</tbody>
</table>

Figure 7. TMOD—Timer/Counter Mode Register

Figure 8. TCON—Timer/Counter Control/Status Register
These peripheral functions allow special hardware to monitor real-time signal interfacing without bothering the CPU. For example, imagine serial data is arriving from one CRT while being transmitted to another, and one timer/counter is tallying high-speed input transitions while the other measures input pulse widths. During all of this the CPU is thinking about something else.

But how does the CPU know when a reception, transmission, count, or pulse is finished? The 8051 programmer can choose from three approaches.

TCON and SCON contain status bits set by the hardware when a timer overflows or a serial port operation is completed. The first technique reads the control register into the accumulator, tests the appropriate bit, and does a conditional branch based on the result. This "polling" scheme (typically a three-instruction sequence though additional instructions to save and restore the accumulator may sometimes be needed) will surely be familiar to programmers used to multi-chip microcomputer systems and peripheral controller chips. This process is rather cumbersome, especially when monitoring multiple peripherals.

As a second approach, the 8051 can perform a conditional branch based on the state of any control or status bit or input pin in a single instruction; a four instruction sequence could poll the four simultaneous happenings mentioned above in just eight microseconds.

Unfortunately, the CPU must still drop what it’s doing to test these bits. A manager cannot do his own work well if he is continuously monitoring his subordinates; they should interrupt him (or her) only when they need attention or guidance. So it is with machines; ideally, the CPU would not have to worry about the peripherals until they require servicing. At that time, it would postpone the background task long enough to handle the appropriate device, then return to the point where it left off.

This is the basis of the third and generally optimal solution, hardware interrupts. The 8051 has five interrupt sources: one from the serial port when a transmission or reception is complete, two from the timers when overflows occur, and two from input pins INTO and INTO. Each source may be independently enabled or disabled to allow polling on some sources or at some times, and each may be classified as high or low priority. A high priority source can interrupt a low priority service routine; the manager's boss can interrupt conferences with subordinates. These options are selected by the interrupt enable and priority control registers, IE and IP (Figures 10 and 11).

Each source has a particular program memory address associated with it (Table 3), starting at 0003H (as in the 8048) and continuing at eight-byte intervals. When an event enabled for interrupts occurs the CPU automatically executes an internal subroutine call to the corresponding address. A user subroutine starting at this location (or jumped to from this location) then performs the instructions to service that particular source. After completing the interrupt service routine, execution returns to the background program.

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Service Routine Starting Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Reset)</td>
<td>0000H</td>
</tr>
<tr>
<td>External 0</td>
<td>0003H</td>
</tr>
<tr>
<td>Timer/Counter 0</td>
<td>000BH</td>
</tr>
<tr>
<td>External 1</td>
<td>0013H</td>
</tr>
<tr>
<td>Timer/Counter 1</td>
<td>001BH</td>
</tr>
<tr>
<td>Serial Port</td>
<td>0023H</td>
</tr>
</tbody>
</table>
### Symbol Position Name and Significance

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA</td>
<td>IE.7</td>
<td>Enable All control bit. Cleared by software to disable all interrupts, independent of the state of IE.4-IE.0.</td>
</tr>
<tr>
<td>---</td>
<td>IE.6</td>
<td>(reserved)</td>
</tr>
<tr>
<td>---</td>
<td>IE.5</td>
<td>(reserved)</td>
</tr>
<tr>
<td>ES</td>
<td>IE.4</td>
<td>Enable Serial port control bit. Set/cleared by software to enable/disable interrupts from TI or RI flags.</td>
</tr>
<tr>
<td>ET1</td>
<td>IE.3</td>
<td>Enable Timer 1 control bit. Set/cleared by software to enable/disable interrupts from timer/counter 1.</td>
</tr>
<tr>
<td>EXI</td>
<td>IE.2</td>
<td>Enable External interrupt 1 control bit. Set/cleared by software to enable/disable interrupts from INT1.</td>
</tr>
<tr>
<td>ET0</td>
<td>IE.1</td>
<td>Enable Timer 0 control bit. Set/cleared by software to enable/disable interrupts from timer counter 0.</td>
</tr>
<tr>
<td>EX0</td>
<td>IE.0</td>
<td>Enable External interrupt 0 control bit. Set/cleared by software to enable/disable interrupts from INT0.</td>
</tr>
</tbody>
</table>

---

**Figure 10. IE—Interrupt Enable Register**

### Symbol Position Name and Significance

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>IP.7</td>
<td>(reserved)</td>
</tr>
<tr>
<td>---</td>
<td>IP.6</td>
<td>(reserved)</td>
</tr>
<tr>
<td>---</td>
<td>IP.5</td>
<td>(reserved)</td>
</tr>
<tr>
<td>PS</td>
<td>IP.4</td>
<td>Serial port Priority control bit. Set/cleared by software to specify high/low priority interrupts for Serial port.</td>
</tr>
<tr>
<td>PTI</td>
<td>IP.3</td>
<td>Timer 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer/counter 1.</td>
</tr>
<tr>
<td>PX1</td>
<td>IP.2</td>
<td>External interrupt 1 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT1.</td>
</tr>
<tr>
<td>PT0</td>
<td>IP.1</td>
<td>Timer 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for timer counter 0.</td>
</tr>
<tr>
<td>PX0</td>
<td>IP.0</td>
<td>External interrupt 0 Priority control bit. Set/cleared by software to specify high/low priority interrupts for INT0.</td>
</tr>
</tbody>
</table>

---

**Figure 11. IP—Interrupt Priority Control Register**
3. INSTRUCTION SET AND ADDRESSING MODES

The 8051 instruction set is extremely regular, in the sense that most instructions can operate with variables from several different physical or logical address spaces. Before getting deeply enmeshed in the instruction set proper, it is important to understand the details of the most common data addressing modes. Whereas Table 4 summarizes the instructions set broken down by functional group, this chapter starts with the addressing mode classes and builds to include the related instructions.

**Data Addressing Modes**

MCS-51 assembly language instructions consist of an operation mnemonic and zero to three operands separated by commas. In two operand instructions the destination is specified first, then the source. Many byte-wide data
operations (such as ADD or MOV) inherently use the accumulator as a source operand and/or to receive the result. For the sake of clarity the letter “A” is specified in the source or destination field in all such instructions. For example, the instruction,

```
ADD A,<source>
```

will add the variable <source> to the accumulator, leaving the sum in the accumulator.

The operand designated “<source>” above may use any of four common logical addressing modes:

- **Register**—one of the working registers in the currently enabled bank.
- **Direct**—an internal RAM location, I/O port, or special-function register.
- **Register-indirect**—an internal RAM location, pointed to by a working register.
- **Immediate data**—an eight-bit constant incorporated into the instruction.

The first three modes provide access to the internal RAM and Hardware Register address spaces, and may therefore be used as source or destination operands; the last mode accesses program memory and may be a source operand only.

(It is hard to show a “typical application” of any instruction without involving instructions not yet described. The following descriptions use only the self-explanatory ADD and MOV instructions to demonstrate how the four addressing modes are specified and used. Subsequent examples will become increasingly complex.)

**Register Addressing**

The 8051 programmer has access to eight “working registers,” numbered R0–R7. The least-significant three-bits of the instruction opcode indicate one register within this logical address space. Thus, a function code and operand address can be combined to form a short (one byte) instruction (Figure 12.a).

The 8051 assembly language indicates register addressing with the symbol Rn (where n is from 0 to 7) or with a symbolic name previously defined as a register by the EQUate or SET directives. (For more information on assembler directives see the Macro Assembler Reference Manual.)

**Example 1—Adding Two Registers Together**

```
ADD A,R0
ADD A,R1
ADD A,R2
ADD A,R3
ADD A,R4
ADD A,R5
ADD A,R6
ADD A,R7
```

There are four such banks of working registers, only one of which is active at a time. Physically, they occupy the first 32 bytes of on-chip data RAM (addresses 0-1FH). PSW bits 4 and 3 determine which bank is active. A hardware reset enables register bank 0; to select a different bank the programmer modifies PSW bits 4 and 3 accordingly.

**Example 2—Selecting Alternate Memory Banks**

```
MOV PSW, #00010000B SELECT BANK 0
```

Register addressing in the 8051 is the same as in the 8048 family, with two enhancements: there are four banks rather than one or two, and 16 instructions (rather than 12) can access them.

**Direct Byte Addressing**

Direct addressing can access any on-chip variable or hardware register. An additional byte appended to the opcode specifies the location to be used (Figure 12.b).

Depending on the highest order bit of the direct address byte, one of two physical memory spaces is selected. When the direct address is between 0 and 127 (00H–7FH) one of the 128 low-order on-chip RAM locations is used. (Future microcomputers based on the MCS-51™ architecture may incorporate more than 128 bytes of on-chip RAM. Even if this is the case, only the low-order 128 bytes will be directly addressable. The remainder would be accessed indirectly or via the stack pointer.)

**Example 3—Adding RAM Location Contents**

```
ADD A, 40H
ADD A, 41H
ADD A, 42H
ADD A, 43H
ADD A, 44H
ADD A, 45H
ADD A, 46H
ADD A, 47H
```

All I/O ports and special function, control, or status registers are assigned addresses between 128 and 255 (80H–0FFH). When the direct address byte is between these limits the corresponding hardware register is accessed. For example, Ports 0 and 1 are assigned direct addresses 80H and 90H, respectively. A complete list is presented in Table 5. Don’t waste your time trying to memorize the addresses in Table 5. Since programs using absolute addresses for function registers would be difficult to write or understand, ASMS1 allows and understands the abbreviations listed instead.

**Example 4—Adding Input Port Data to Output Port Data**

```
MOV A, PO
ADD A, PO
```

Direct addressing allows all special-function registers in the 8051 to be read, written, or used as instruction operands. In general, this is the _only_ method used for accessing I/O ports and special-function registers. If direct addressing is used with special-function register addresses other than those listed, the result of the instruction is undefined.

AFN-01502A-1B
The 8048 does not have or need any generalized direct addressing mode, since there are only five special registers (BUS, P1, P2, PSW, & T) rather than twenty. Instead, 16 special 8048 opcodes control output bits or read or write each register to the accumulator. These functions are all subsumed by four of the 27 direct addressing instructions of the 8051.

Table 5. 8051 Hardware Register Direct Addresses

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>80H*</td>
<td>Port 0</td>
</tr>
<tr>
<td>SP</td>
<td>81H</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>DPL</td>
<td>82H</td>
<td>Data Pointer (Low)</td>
</tr>
<tr>
<td>DPH</td>
<td>83H</td>
<td>Data Pointer (High)</td>
</tr>
<tr>
<td>TCON</td>
<td>88H*</td>
<td>Timer register</td>
</tr>
<tr>
<td>TMOD</td>
<td>89H</td>
<td>Timer Mode register</td>
</tr>
<tr>
<td>I1.0</td>
<td>8AH</td>
<td>Timer 0 Low byte</td>
</tr>
<tr>
<td>I1.1</td>
<td>8BH</td>
<td>Timer 1 Low byte</td>
</tr>
<tr>
<td>I1.0</td>
<td>8CH</td>
<td>Timer 0 High byte</td>
</tr>
<tr>
<td>I1.1</td>
<td>8DH</td>
<td>Timer 1 High byte</td>
</tr>
<tr>
<td>PI</td>
<td>90H*</td>
<td>Port 1</td>
</tr>
<tr>
<td>SCON</td>
<td>91H</td>
<td>Serial Port Control register</td>
</tr>
<tr>
<td>SBUF</td>
<td>92H</td>
<td>Serial Port Data Buffer</td>
</tr>
<tr>
<td>P2</td>
<td>00H*</td>
<td>Port 2</td>
</tr>
<tr>
<td>IE</td>
<td>00H</td>
<td>Interrupt Enable register</td>
</tr>
<tr>
<td>P3</td>
<td>01H</td>
<td>Port 3</td>
</tr>
<tr>
<td>IP</td>
<td>02H</td>
<td>Interrupt Priority register</td>
</tr>
<tr>
<td>PSW</td>
<td>03H</td>
<td>Program Status Word</td>
</tr>
<tr>
<td>ACC</td>
<td>00H</td>
<td>Accumulator (direct address)</td>
</tr>
<tr>
<td>B</td>
<td>01H*</td>
<td>B register</td>
</tr>
</tbody>
</table>

* = bit addressable register

Register-Indirect Addressing

How can you handle variables whose locations in RAM are determined, computed, or modified while the program is running? This situation arises when manipulating sequential memory locations, indexed entries within tables in RAM, and multiple precision or string operations. Register or Direct addressing cannot be used, since their operand addresses are fixed at assembly time.

The 8051 solution is “register-­indirect RAM addressing.” R0 and R1 of each register bank may operate as index or pointer registers, their contents indicating an address into RAM. The internal RAM location so addressed is the actual operand used. The least significant bit of the instruction opcode determines which register is used as the “pointer” (Figure 12.c).

In the 8051 assembly language, register-­indirect addressing is represented by a commercial “at” sign (“@”) preceding R0, R1, or a symbol defined by the user to be equal to R0 or R1.

Example 5 — Indirect Addressing

```
.PINDXR ADD CONTENTS OF MEMORY LOCATION
  .ADDED BY REGISTER I
  .TO CONTENTS OF RAM LOCATION
  .ADDED BY REGISTER 0

.IINDXR MOV A, @R0
  ADD A, R0
  MOV R0, A
```

Indirect addressing on the 8051 is the same as in the 8048 family, except that all eight bits of the pointer register contents are significant; if the contents point to a non-existent memory location (i.e., an address greater than 7FH on the 8051) the result of the instruction is undefined. (Future microcomputers based on the MCS-51™ architecture could implement additional memory in the on-chip RAM logical address space at locations above 7FH.) The 8051 uses register-­indirect addressing for five new instructions plus the 13 on the 8048.

Immediate Addressing

When a source operand is a constant rather than a variable (i.e., the instruction uses a value known at assembly time), then the constant can be incorporated into the instruction. An additional instruction byte specifies the value used (Figure 12.d).

The value used is fixed at the time of ROM manufacture or EPROM programming and may not be altered during program execution. In the assembly language immediate operands are preceded by a number sign (“#”). The operand may be either a numeric string, a symbolic variable, or an arithmetic expression using constants.

Example 6 — Adding Constants Using Immediate Addressing

```
.IIMADR ADD THE CONSTANT 12 (DECIMAL)
  TO THE CONSTANT 34 (DECIMAL)
  LEAVE SUM IN ACCUMULATOR

.IIMADR MOV A, #12
  ADD A, #34
```

The preceding example was included for consistency; it has little practical value. Instead, ASM51 could compute the sum of two constants at assembly time.

Example 7 — Adding Constants Using ASM51 Capabilities

```
.ASM51 LOAD ACC WITH THE SUM OF
  THE CONSTANT 12 (DECIMAL) AND
  THE CONSTANT 34 (DECIMAL)

ASM51 MOV A, #12+34
```

Figure 12. Data Addressing Machine Code Formats

a) Register Addressing:

```
opcode
  n
  R n
ADD A, R
```

b) Direct Addressing:

```
opcode
direct address
ADD A, direct
```

c) Register-­indirect Addressing:

```
opcode
ADD A, @R
```

d) Immediate Addressing:

```
opcode data
ADD A, data
```
Symbolic Addressing of Variables

The above examples all demonstrated the use of the four data-addressing modes in two-operand instructions (MOV, ADD) which use the accumulator as one operand. The operations ADDC, SUBB, ANL, ORL, and XRL (all to be discussed later) could be substituted for ADD in each example. The first three modes may be also be used for the XCH operation or, in combination with the Immediate Addressing mode (and an additional byte), loaded with a constant. The one-operand instructions INC and DEC, DJNZ, and CJNE may all operate on the accumulator, or may specify the Register, Direct, and Register-indirect addressing modes. Exception: as in the 8048, DJNZ cannot use the accumulator or indirect addressing. (The PUSH and POP operations cannot inherently address the accumulator as a special register either. However, all three can directly address the accumulator as one of the twenty special-function registers by putting the symbol “ACC” in the operand field.)

Advantages of Symbolic Addressing

Like most assembly or higher-level programming languages, ASM51 allows instructions or variables to be given appropriate, user-defined symbolic names. This is done for instruction lines by putting a label followed by a colon (“:”) before the instruction proper, as in the above examples. Such symbols must start with an alphabetic character (remember what distinguished BACH from "Bach"), and may include any combination of letters, numbers, question marks (“?”) and underscores (“_”). For very long names only the first 31 characters are relevant.

Assembly language programs may intermix upper- and lower-case letters arbitrarily, but ASM51 converts both to upper-case. For example, ASM51 will internally process an “i” for an “I” and, of course, “A_TOOTH” for “a_tooth.”

The underscore character makes symbols easier to read and can eliminate potential ambiguity (as in the label for a subroutine to switch two entries on a stack, “S_EXCHANGE”). The underscore is significant, and would distinguish between otherwise-identical character strings.

ASM51 allows all variables (registers, ports, internal or external RAM addresses, constants, etc.) to be assigned labels according to these rules with the EQUate or SET directives.

Example 8 - Symbolic Addressing of Variables Defined as RAM Locations

```
VAR_0 SET 20H
VAR_1 SET 21H
SYMB_1 ADD CONTENTS OF VAR_1 TO CONTENTS OF VAR_0
SYMB_1 MOV A, VAR_0
ADD A, VAR_1
MOV VAR_0 A
```

Notice from Table 4 that the MCS-51™ instruction set has relatively few instruction mnemonics (abbreviations) for the programmer to memorize. Different data types or addressing modes are determined by the operands specified, rather than variations on the mnemonic. For example, the mnemonic “MOV” is used by 18 different instructions to operate on three data types (bit, byte, and address). The fifteen versions which move byte variables between the logical address spaces are diagrammed in Figure 13. Each arrow shows the direction of transfer from source to destination.

Notice also that for most instructions allowing register addressing there is a corresponding direct addressing instruction and vice versa. This lets the programmer begin writing 8051 programs as if (s)he has access to 128 different registers. When the program has evolved to the point where the programmer has a fairly accurate idea how often each variable is used, he may allocate the working registers in each bank to the most “popular” variables. (The assembly cross-reference option will show exactly how often and where each symbol is referenced.)

If symbolic addressing is used in writing the source program only the lines containing the symbol definition will need to be changed; the assembler will produce the appropriate instructions even though the rest of the program is left untouched. Editing only the first two lines of Example 8 will shrink the six-byte code segment produced in half.

How are instruction sets “counted”? There is no standard practice; different people assessing the same CPU using different conventions may arrive at different totals.

Each operation is then broken down according to the different addressing modes (or combinations of addressing modes) it can accommodate. The “CLR” mnemonic is used by two instructions with respect to bit variables (“CLR C” and “CLR bit”) and once (“CLR A”) with regards to bytes. This expansion yields the 111 separate instructions of Table 4.

The method used for the MCS-51® instruction set first breaks it down into “operations”: a basic function applied to a single data type. For example, the four versions of the ADD instruction are grouped to form one operation — addition of eight-bit variables. The six forms of the ANL instruction for byte variables make up a different operation; the two forms of ANL which operate on bits are considered still another. The MOV mnemonic is used by three different operation classes, depending on whether bit, byte, or 16-bit values are affected. Using this terminology the 8051 can perform 51 different operations.
Example 9 — Redeclaring Example 8 Symbols as Registers

```
VAR_0 SET RO
VAR_1 SET R1
.SYM_2 ADD CONTENTS OF VAR_1 TO CONTENTS OF VAR_0
SYM_2 MOV A:VAR_0
ADD A:VAR_1
MOV VAR_0
```

**Arithmetic Instruction Usage — ADD, ADDC, SUBB and DA**

The ADD instruction adds a byte variable with the accumulator, leaving the result in the accumulator. The carry flag is set if there is an overflow from bit 7 and cleared otherwise. The AC flag is set to the carry-out from bit 3 for use by the DA instruction described later. ADDC adds the previous contents of the carry flag with the two byte variables, but otherwise is the same as ADD.

The SUBB (subtract with borrow) instruction subtracts the byte variable indicated and the contents of the carry flag together from the accumulator, and puts the result back in the accumulator. The carry flag serves as a "Borrow Required" flag during subtraction operations; when a greater value is subtracted from a lesser value (as in subtracting 5 from 1) requiring a borrow into the highest order bit, the carry flag is set; otherwise it is cleared.

When performing signed binary arithmetic, certain combinations of input variables can produce results which seem to violate the Laws of Mathematics. For example, adding 7FH (127) to itself produces a sum of 0FEH, which is the two's complement representation of -2 (refer back to Table 2). In “normal” arithmetic, two positive values can’t have a negative sum. Similarly, it is normally impossible to subtract a positive value from a negative value and leave a positive result — but in two’s complement there are instances where this too may happen. Fundamentally, such anomalies occur when the magnitude of the resulting value is too great to “fit” into the seven bits allowed for it; there is no one-byte two’s complement representation for 254, the true sum of 127 and 127.

Decimal addition is possible by using the DA instruction in conjunction with ADD and/or ADDC. The eight-bit binary value in the accumulator resulting from an earlier addition of two variables (each a packed BCD digit-pair) is adjusted to form two BCD digits of four bits each. If the contents of the accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag had been set, six is added to the accumulator producing the proper BCD digit in the low-order nibble. (This addition might itself set — but would not clear — the carry flag.) If the carry flag is set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these bits are incremented by six. The carry flag is left set if originally set or if either addition of six produces a carry out of the highest-order bit, indicating the sum of the original two BCD variables is greater than or equal to decimal 100.

The MCS-51 processors detect whether these situations occur and indicate such errors with the OV flag. (OV may be tested with the conditional jump instructions JB and JNB, described under the Boolean Processor chapter.) At a hardware level, OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6. When adding signed integers this indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands; on SUBB this indicates a negative result after subtracting a negative number from a positive number, or a positive result when a positive number is subtracted from a negative number.

The ADDC and SUBB instructions incorporate the previous state of the carry (borrow) flag to allow multiple precision calculations by repeating the operation with successively higher-order operand bytes. In either case, the carry must be cleared before the first iteration.

If the input data for a multiple precision operation is an unsigned string of integers, upon completion the carry flag will be set if an overflow (for ADDC) or underflow (for SUBB) occurs. With two's complement signed data (i.e., if the most significant bit of the original input data indicates the sign of the string), the overflow flag will be set if overflow or underflow occurred.

Example 10 — String Subtraction with Signed Overflow Detection

```
.SUBSTR SUBTRACT STRING INDICATED BY RI:

FROM STRING INDICATED BY RO TO:

PRECISION INDICATED BY BV:

CHECK FOR SIGNED UNDERFLOW WHEN DONE:

.SUBSTR CLR C .BORROW=O

SUBS1 CLR C .BORROW=O

SUBSTR CLR C .BORROW=O

INC RO .BUMP POINTERS

INC RI .BUMP POINTERS

NEXT DV: SUBST .LOOP AS NEEDED

WHEN DONE, TEST IF OVERFLOW OCCURRED

LAST ITERATION OF LOOP:

JNB OV:OV OK .OVERFLOW RECOVERY ROUTINE

OV_OK RET .RETURN
```

Figure 13. Road map for moving data bytes
Example 11 — Two Byte Decimal Add with Registers and Constants

```
.BCDADD ADD THE CONSTANT 1.234 (DECIMAL) TO THE
CONTENTS OF REGISTER PAIR (R2/R3)
(Already A + B BCD-DIGIT VARIABLE)

.BCDADD MOV A,R2
ADD A,R3H
DA A
MOV R2,A
MOV A,R3
ASDC A,R1H
DA A
MOV R3,A
RET
```

Multiplication and Division

The instruction "MUL AB" multiplies the unsigned eight-bit integer values held in the accumulator and B-registers. The low-order byte of the sixteen-bit product is left in the accumulator, the higher-order byte in B. If the high-order eight-bits of the product are all zero the overflow flag is cleared; otherwise it is set. The programmer can poll OV to determine when the B register is non-zero and must be processed.

"DIV AB" divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in the B-register. The integer part of the quotient is returned in the accumulator; the remainder in the B-register. If the B-register originally contained 00H then the overflow flag will be set to indicate a division error, and the values returned will be undefined. Otherwise OV is cleared.

The divide instruction is also useful for purposes such as radix conversion or separating bit fields of the accumulator. A short subroutine can convert an eight-bit unsigned binary integer in the accumulator (between 0 & 255) to a three-digit (two byte) BCD representation. The hundred's digit is returned in one register (HUND) and the ten's and one's digits returned as packed BCD in another (TENONE).

Example 12 — Use of DIV Instruction for Radix Conversion

```
.BINBCD CONVERT 8-BIT BINARY VARIABLE IN ACC
TO 3-DIGIT PACKED BCD FORMAT.
Hundreds' Place Left in Variable 'HUND',
Tens' and Ones' Places in 'TENONE'

HUND EQU 21H
TENONE EQU 22H

.BINBCD MOV B, #100 ; Divide by 100 to
DIV AB ; Determine number of hundreds
MOV HUND,A
MOV A, #10 ; Divide remainder by 10 to
DIV AB ; Determine # of tens left
MOV A, #1 ; Determine # of ones left
DIV AB
SWAP A
ADD A,B ; Pack BCD digits in ACC
MOV TENONE,A
RET
```

The divide instruction can also separate eight bytes of data in the accumulator into sub-fields. For example, packed BCD data may be separated into two nibbles by dividing the data by 16, leaving the high-nibble in the accumulator and the low-order nibble (remainder) in B. The two digits may then be operated on individually or in conjunction with each other. This example receives two packed BCD digits in the accumulator and returns the product of the two individual digits in packed BCD format in the accumulator.

Example 13 — Implementing a BCD Multiply Using MPY and DIV

```
.MULBCD UNPACK TWO BCD DIGITS RECEIVED IN ACC.
FIND THEIR PRODUCT, AND RETURN PRODUCT
IN PACKED BCD FORMAT IN ACC

.MULBCD MOV B, #10H ; Divide input by 10
DIV AB ; A & B hold separated digits
.MOV A, #00H ; A holds product in binary format (0 - 63H)
.MOV B, #10H ; Divide product by 10
DIV AB ; A holds # of tens, B holds remainder
.SWAP A
.ORL A,B ; Pack digits
.RET
```

Logical Byte Operations — ANL, ORL, XRL

The instructions ANL, ORL, and XRL perform the logical functions AND, OR, and/or Exclusive-OR on the two byte variables indicated, leaving the results in the first. No flags are affected. (A word to the wise — do not vocalize the first two mnemonics in mixed company.)

These operations may use all the same addressing modes as the arithmetics (ADD, etc.) but unlike the arithmetics, they are not restricted to operating on the accumulator. Directly addressed bytes may be used as the destination with either the accumulator or a constant as the source. These instructions are useful for clearing (ANL), setting (ORL), or complementing (XRL) one or more bits in a RAM, output ports, or control registers. The pattern of bits to be affected is indicated by a suitable mask byte. Use immediate addressing when the pattern to be affected is known at assembly time (Figure 14); use the accumulator versions when the pattern is computed at run-time.

I/O ports are often used for parallel data in formats other than simple eight-bit bytes. For example, the low-order five bits of port 1 may output an alphabetic character code (hopefully) without disturbing bits 7-5. This can be a simple two-step process. First, clear the low-order five pins with an ANL instruction; then set those pins corresponding to ones in the accumulator. (This example assumes the three high-order bits of the accumulator are originally zero.)

Example 14 — Reconfiguring Port Size with Logical Byte Instructions

```
OUT_FX ANL P1, #1100000B ; Clear bits P1.4 - P1.0
ORL P1,A ; Set 1 corresponding to set ACC
.RET
```

Figure 14. Instruction Pattern for Logical Operation
Special Addressing Modes

<table>
<thead>
<tr>
<th>opcode</th>
<th>direct address</th>
<th>mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANL</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;data&gt;</td>
<td></td>
</tr>
</tbody>
</table>
In this example, low-order bits remaining high may "glitch" low for one machine cycle. If this is undesirable, use a slightly different approach. First, set all pins corresponding to accumulator one bits, then clear the pins corresponding to zeroes in low-order accumulator bits. Not all bits will change from original to final state at the same instant, but no bit makes an intermediate transition.

Example 15 — Reconfiguring I/O Port Size without Glitching

Program Control — Jumps, Calls, Returns

Whereas the 8048 only has a single form of the simple jump instruction, the 8051 has three. Each causes the program to unconditionally jump to some other address. They differ in how the machine code represents the destination address.

LJMP (Long Jump) encodes a sixteen-bit address in the second and third instruction bytes (Figure 15.a); the destination may be anywhere in the 64 Kilobyte program memory address space.

The two-byte AJMP (Absolute Jump) instruction encodes its destination using the same format as the 8048: address bits 10 through 8 form a three bit field in the opcode and address bits 7 through 0 form the second byte (Figure 15.b). Address bits 15-12 are unchanged from the (incremented) contents of the P.C., so AJMP can only be used when the destination is known to be within the same 2K memory block. (Otherwise ASM51 will point out the error.)

A different two-byte jump instruction is legal with any nearby destination, regardless of memory block boundaries or "pages." SJMP (Short Jump) encodes the destination with a program counter-relative address in the second byte (Figure 15.c). The CPU calculates the destination at run-time by adding the signed eight-bit displacement value to the incremented P.C. Negative offset values will cause jumps up to 128 bytes backwards; positive values up to 127 bytes forwards. (SJMP with 00H in the machine code offset byte will proceed with the following instruction).

In keeping with the 8051 assembly language goal of minimizing the number of instruction mnemonics, there is a "generic" form of the three jump instructions. ASM51 recognizes the mnemonic JMP as a "pseudo-instruction," translating it into the machine instructions LJMP, AJMP, or SJMP, depending on the destination address.

Like SJMP, all conditional jump instructions use relative addressing. JZ (Jump if Zero) and JNZ (Jump if Not Zero) monitor the state of the accumulator as implied by their names, while JC (Jump on Carry) and JNC (Jump on No Carry) test whether or not the carry flag is set. All four are two-byte instructions, with the same format as Figure 15.c. JB (Jump on Bit), JNB (Jump on No Bit) and JBC (Jump on Bit then Clear Bit) can test any status bit or input pin with a three byte instruction; the second byte specifies which bit to test and the third gives the relative offset value.

There are two subroutine-call instructions, LCALL (Long Call) and ACALL (Absolute Call). Each increments the P.C. to the first byte of the following instruction, then pushes it onto the stack (low byte first). Saving both bytes increments the stack pointer by two. The subroutine's starting address is encoded in the same ways as LJMP and AJMP. The generic form of the call operation is the mnemonic CALL, which ASM51 will translate into LCALL or ACALL as appropriate.

The return instruction RET pops the high- and low-order bytes of the program counter successively from the stack, decrementing the stack pointer by two. Program execution continues at the address previously pushed; the first byte of the instruction immediately following the call.

When an interrupt request is recognized by the 8051 hardware, two things happen. Program control is automatically "vectored" to one of the interrupt service routine starting addresses by, in effect, forcing the CPU to process an LCALL instead of the next instruction. This automatically stores the return address on the stack. (Unlike the 8048, no status information is automatically saved.) Secondly, the interrupt logic is disabled from accepting any other interrupts from the same or lower priority. After completing the interrupt service routine, executing an RETI (Return from Interrupt) instruction will return execution to the point where the background program was interrupted — just like RET — while restoring the interrupt logic to its previous state.

Figure 15. Jump Instruction Machine Code Formats

| a) Long Jump (LJMP addr16) | opcode | addr15 — addr8 | addr7 — addr0 |
| b) Absolute Jump (AJMP addr11) | | | |
| | add10—add8 | opcode | addr7 — addr0 |
| c) Short Jump (SJMP rel) | | | |
| | opcode | | relative offset |

2-21

AFN-015020A-23
Operate-and-branch instructions — CJNE, DJNZ

Two groups of instructions combine a byte operation with a conditional jump based on the results. CJNE (Compare and Jump if Not Equal) compares two byte operands and executes a jump if they disagree. The carry flag is set following the rules for subtraction; if the unsigned integer value of the first operand is less than that of the second it is set; otherwise, it is cleared. However, neither operand is modified.

The CJNE instruction provides, in effect, a one-instruction “case” statement. This instruction may be executed repeatedly, comparing the code variable to a list of “special case” values; the code segment following the instruction (up to the destination label) will be executed only if the operand matches. Comparing the accumulator or a register to a series of constants is a convenient way to check for special handling or error conditions; if none of the cases match the program will continue with “normal” processing.

A typical example might be a word processing device which receives ASCII characters through the serial port and drives a thermal hard-copy printer. A standard routine translates “printing” characters to bit patterns, but control characters (DEL, CR, LF, BEL, ESC or SP) must invoke corresponding special routines. Any other character with an ASCII code less than 20H should be translated into the <NUL> value, 00H, and processed with the printing characters.

Example 16 — Case Statements Using CJNE

```
Example 18 — Use of the Stack for Status Saving on Interrupts
```

DJNZ (Decrement and Jump if Not Zero) decrements the register or direct address indicated and jumps if the result is not zero, without affecting any flags. This provides a simple means for executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. For example, a 99-ussec software delay loop can be added to code forcing an 1/O pin low with only two instructions.

Example 17 — Inserting a Software Delay with DJNZ

```
The dollar sign in this example is a special character meaning “the address of this instruction.” It is useful in eliminating instruction labels on the same or adjacent source lines. CJNE and DJNZ (like all conditional jumps) use program-counter relative addressing for the destination address.

Stack Operations — PUSH, POP

The PUSH instruction increments the stack pointer by one, then transfers the contents of the single byte variable indicated (direct addressing only) into the internal RAM location addressed by the stack pointer. Conversely, POP copies the contents of the internal RAM location addressed by the stack pointer to the byte variable indicated, then decrements the stack pointer by one. (Stack Addressing follows the same rules, and addresses the same locations as Register-indirect. Future microcomputers based on the MCS-51" CPU could have up to 256 bytes of RAM for the stack.)

Interrupt service routines must not change any variable or hardware registers modified by the main program, or else the program may not resume correctly. (Such a change might look like a spontaneous random error.) Resources used or altered by the service routine (Accumulator, PSW, etc.) must be saved and restored to their previous value before returning from the service routine. PUSH and POP provide an efficient and convenient way to save register states on the stack.

If the SP register held 1FH when the interrupt was detected, then while the service routine was in progress the stack would hold the registers shown in Figure 16; SP would contain 26H.

The example shows the most general situation; if the service routine doesn’t alter the B-register and data pointer, for example, the instructions saving and restoring those registers would not be necessary.

The stack may also pass parameters to and from subroutines. The subroutine can indirectly address the parameters derived from the contents of the stack pointer.
One advantage here is simplicity. Variables need not be allocated for specific parameters. A potentially large number of parameters may be passed, and different calling programs may use different techniques for determining or handling the variables.

For example, the following subroutine reads out a parameter stored on the stack by the calling program, uses the low order bits to access a local look-up table holding bit patterns for driving the coils of a four phase stepper motor, and stores the appropriate bit pattern back in the same position on the stack before returning. The accumulator contents are left unchanged.

Example 19 — Passing Variable Parameters to Subroutines Using the Stack

```assembly
NXTPOS MOV @R0, SP 
DEC R0 : ACCESS LOCATION PARAMETER PUSHED INTO 
DEC R0 
XCH A, R0 : READ INPUT PARAMETER AND SAVE 
XCH A, ACC : ACCUMULATOR 
ANL A, #03H : MARK ALL BUT LOW-ORDER TWO BITS 
ADD A, #30 : ALLOW FOR OFFSET FROM MOVC TO TABLE 
MOVC A, @A+DPTR : READ LOOK-UP TABLE ENTRY 
XCH A, R0 : PASS BACK TRANSLATED VALUE AND RESTORE 
XCH A, ACC 
RET : RETURN TO BACKGROUND PROGRAM 
```

The background program may reach this subroutine with several different calling sequences, all of which PUSH a value before calling the routine and POP the result after. A motor on Port 1 may be initialized by placing the desired position (zero) on the stack before calling the subroutine and outputting the results directly to a port afterwards.

Example 20 — Sending and Receiving Data Parameters Via the Stack

```assembly
CLR A 
PUSH ACC 
CALL NXTPOS 
PDP P1 
```

If the position of the motor is determined by the contents of variable POSM1 (a byte in internal RAM) and the position of a second motor on Port 2 is determined by the data input to the low-order nibble of Port 2, a six-instruction sequence could update them both.

Example 21 — Loading and Unloading Stack Direct from I/O Ports

```assembly
POSM EQU 51 
PUSH POSM 
CALL NXTPOS 
PDP P1 
PUSH P2 
CALL NXTPOS 
PDP P2 
```

Data Pointer and Table Look-up instructions — MOVC, INC, MOV, JMP

The data pointer can be loaded with a 16-bit value using the instruction MOVC A,@A+DPTR and MOVC A,@A+PC) read into the accumulator bytes of data from the program memory logical address space. Both use a form of indexed addressing: the former adds the unsigned eight-bit accumulator contents with the sixteen-bit data pointer register, and uses the resulting sum as the address from which the byte is fetched. A sixteen-bit addition is performed; a carry-out from the low-order byte will carry into the high-order byte. Neither instruction affects any flags.

The MOVC (Move Constant) instructions (MOVC A,A+DPTR and MOVC A,A+PC) read into the accumulator bytes of data from the program memory logical address space. Both use a form of indexed addressing: the former adds the unsigned eight-bit accumulator contents with the sixteen-bit data pointer register, and uses the resulting sum as the address from which the byte is fetched. A sixteen-bit addition is performed; a carry-out from the low-order eight bits may propagate through higher-order bits, but the contents of the DPTR are not altered. The latter form uses the incremented program counter as the “base” value instead of the DPTR (figure 17). Again, neither version affects the flags.

```assembly
a) MOVC A,A+PC 
  (LOCAL TABLE LOOK-UP) 
  16-BIT PC 
  8-BIT ACC 

b) MOVC A,A+DPTR 
  (GLOBAL TABLE LOOK-UP) 
  16-BIT DPTR 
  8-BIT ACC 

c) JMP A,A+DPTR 
  (GLOBAL INDIRECT JUMP) 
  16-BIT DPTR 
  8-BIT ACC 
```

Figure 17. Operation of MOVC instructions
Each can be part of a three step sequence to access look-up tables in ROM. To use the DPTR-relative version, load the Data Pointer with the starting address of a look-up table; load the accumulator with (or compute) the index of the entry desired; and execute MOVC A,@A+DPTR. Unlike the similar MOV3 instructions in the 8048, the table may be located anywhere in program memory. The data pointer may be loaded with a constant for short tables. Or to allow more complicated data structures, or tables with more than 256 entries, the values for DPH and DPL may be computed or modified with the standard arithmetic instruction set.

The PC-relative version has the advantage of not affecting the data pointer. Again, a look-up sequence takes three steps: load the accumulator with the index; compensate for the offset from the look-up instruction to the start of the table by adding the number of bytes separating them to the accumulator; then execute the MOVC A,@A+PC instruction.

Let's look at a non-trivial situation where this instruction would be used. Some applications store large multidimensional look-up tables of dot matrix patterns, non-linear calibration parameters, and so on in a linear (one-dimensional) vector in program memory. To retrieve data from the tables, variables representing matrix indices must be converted to the desired entry's memory address. For a matrix of dimensions (MDIMEN x NDIMEN) starting at address BASE and respective indices INDEX I and INDEX J, the address of element (INDEX I, INDEX J) is determined by the formula,

Entry Address = BASE + ((INDEX I x NDIMEN) + INDEX J)

The code shown below can access any array with less than 255 entries (i.e., an 11x21 array with 231 elements). The table entries are defined using the Data Byte ("DB") directive, and will be contained in the assembly object code as part of the accessing subroutine itself.

Example 22 — Use of MPY and Data Pointer Instructions to Access Entries from a Multi-dimensional Look-Up Table in ROM

There are several different means for branching to sections of code determined or selected at run time. (The single destination addresses incorporated into conditional and unconditional jumps are, of course, determined at assembly time). Each has advantages for different applications.

The most common is an N-way conditional jump based on some variable, with all of the potential destinations known at assembly time. One of a number of small routines is selected according to the value of an index variable determined while the program is running. The most efficient way to solve this problem is with the MOVC and an indirect jump instruction, using a short table of one byte offset values in ROM to indicate the relative starting addresses of the several routines.

JMP @A+DPTR is an instruction which performs an indirect jump to an address determined during program execution. The instruction adds the eight-bit unsigned accumulator contents with the contents of the sixteen-bit data pointer, just like MOVC A,@A+DPTR. The resulting sum is loaded into the program counter and is used as the address for subsequent instruction fetches. Again, a sixteen-bit addition is performed; a carry out from the low-order eight bits may propagate through the higher-order bits. In this case, neither the accumulator contents nor the data pointer is altered.

The example subroutine below reads a byte of RAM into the accumulator from one of four alternate address spaces, as selected by the contents of the variable MEMSEL. The address of the byte to be read is determined by the contents of R0 (and optionally R1). It might find use in a printing terminal application, where four different model printers all use the same ROM code but use different types and sizes of buffer memory for different speeds and options.

Example 23 — N-Way Branch and Computed Jump Instructions via JMP @ ADPTR

Note that this approach is suitable whenever the size of jump table plus the length of the alternate routines is less than 256 bytes. The jump table and routines may be located anywhere in program memory, independent of 256-byte program memory pages.
For applications where up to 128 destinations must be selected, all of which reside in the same 2K page of program memory which may be reached by the two-byte absolute jump instructions, the following technique may be used. In the above mentioned printing terminal example, this sequence could “parse” 128 different codes for ASCII characters arriving via the 8051 serial port.

**Example 24 — N-Way Branch with 128 Optional Destinations**

```plaintext
<table>
<thead>
<tr>
<th>OPTION</th>
<th>EQU</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP120</td>
<td>MOV</td>
<td>A, option</td>
</tr>
<tr>
<td></td>
<td>JMP</td>
<td>A, option</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>BPH, [DSPL]</td>
</tr>
<tr>
<td>JMP</td>
<td>BPH, 00H</td>
<td>Jump into jump table</td>
</tr>
<tr>
<td>DSPL</td>
<td>A, JMP</td>
<td>PROC000</td>
</tr>
<tr>
<td>A, JMP</td>
<td>PROC001</td>
<td>.JMP instructions</td>
</tr>
<tr>
<td>A, JMP</td>
<td>PROC002</td>
<td></td>
</tr>
<tr>
<td>A, JMP</td>
<td>PROC07C</td>
<td></td>
</tr>
<tr>
<td>A, JMP</td>
<td>PROC07F</td>
<td></td>
</tr>
</tbody>
</table>
```

The destinations in the jump table (PROC000-PROC07F) are not all necessarily unique routines. A large number of special control codes could each be processed with their own unique routine, with the remaining printing characters all causing a branch to a common routine for entering the character into the output queue.

In those rare situations where even 128 options are insufficient, or where the destination routines may cross a 2K page boundary, the above approach may be modified slightly as shown below.

**Example 25 — 256-Way Branch Using Address Look-Up Tables**

```plaintext
<table>
<thead>
<tr>
<th>RETMP</th>
<th>EQU</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP200</td>
<td>MOV</td>
<td>DPTR, MDSTBL .First entry in table of addresses</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>A, option</td>
</tr>
<tr>
<td>CLR</td>
<td>A</td>
<td>.Multiply by 2 for 2 byte jump table</td>
</tr>
<tr>
<td>INC</td>
<td>LOW128</td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>DPH</td>
<td></td>
</tr>
<tr>
<td>LOW128</td>
<td>MOV</td>
<td>RTMP, D</td>
</tr>
<tr>
<td>MOV</td>
<td>A, DPTR</td>
<td>.Read low byte from jump table</td>
</tr>
<tr>
<td>INC</td>
<td>A</td>
<td>.Add to previous increment</td>
</tr>
<tr>
<td>INC</td>
<td>A, DPTR</td>
<td>.Get low order byte from table</td>
</tr>
<tr>
<td>PUSH</td>
<td>ACC</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>A, RTMP</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>A, DPTR</td>
<td>.Get high-order byte from table</td>
</tr>
<tr>
<td>PUSH</td>
<td>ACC</td>
<td></td>
</tr>
<tr>
<td>THE TWO ACC PUSHES HAVE PRODUCED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A RETURN ADDRESS ON THE STACK WHICH CORRESPONDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TO THE DESIRED STARTING ADDRESS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT MAY BE REACHED BY POPPING THE STACK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTO THE PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDTB</td>
<td>SW</td>
<td>PROC000 UP TO 256 CONSECUTIVE DATA</td>
</tr>
<tr>
<td></td>
<td>SW</td>
<td>PROC001 WORDS INDICATING STARTING ADDRESSES</td>
</tr>
<tr>
<td></td>
<td>SW</td>
<td>PROC0FF</td>
</tr>
<tr>
<td></td>
<td>DW</td>
<td>DUMMY CODE ADDRESS DEFINITIONS NEEDED BY ABOVE</td>
</tr>
<tr>
<td></td>
<td>. TWO EXAMPLES</td>
<td></td>
</tr>
<tr>
<td>PROC000</td>
<td>NDR</td>
<td></td>
</tr>
<tr>
<td>PROC001</td>
<td>NDR</td>
<td></td>
</tr>
<tr>
<td>PROC002</td>
<td>NDR</td>
<td></td>
</tr>
<tr>
<td>PROC07C</td>
<td>NDR</td>
<td></td>
</tr>
<tr>
<td>PROC07F</td>
<td>NDR</td>
<td></td>
</tr>
<tr>
<td>PROC0FF</td>
<td>NDR</td>
<td></td>
</tr>
</tbody>
</table>
```

Prior to the introduction of the MCS-51™ family, nice number-crunchers made bad bit-bangers and vice versa. The 8051 is the industry’s first single-chip microcomputer designed to crunch and bang. (In some circles, the latter technique is also referred to as “bit-twiddling”. Either is correct.)

**Direct Bit Addressing**

A number of instructions operate on Boolean (one-bit) variables, using a direct bit addressing mode comparable to direct byte addressing. An additional byte appended to the opcode specifies the Boolean variable, I O pin, or control bit used. The state of any of these bits may be tested for “true” or “false” with the conditional branch instructions JB (Jump on Bit) and JNB (Jump on Not Bit). The JBC (Jump on Bit and Clear) instruction combines a test-for-true with an unconditional clear.

As in direct byte addressing, bit 7 of the address byte switches between two physical address spaces. Values between 0 and 127 (00H-7FH) define bits in internal RAM locations 20H to 2FH (Figure 18a); address bytes between 128 and 255 (80H-0FFH) define bits in the 2 x “special-function” register address space (Figure 18b). If no 2 x “special-function” register corresponds to the direct bit address used the result of the instruction is undefined.

Bits so addressed have many wonderful properties. They may be set, cleared, or complemented with the two byte instructions SETB, CLR, or CPL. Bits may be moved to and from the carry flag with MOV. The logical ANL and ORL functions may be performed between the carry and either the addressed bit or its complement.

**Bit Manipulation Instructions — MOV**

The “MOV” mnemonic can be used to load an addressable bit into the carry flag (“MOV C, bit”) or to copy the state of the carry to such a bit (“MOV bit, C”). These instructions are often used for implementing serial I O algorithms via software or to adapt the standard I O port structure.

It is sometimes desirable to “re-arrange” the order of I O pins because of considerations in laying out printed circuit boards. When interfac ing the 8051 to an immediately adjacent device with “weighted” input pins, such as keyboard column decoder, the corresponding pins are likely to be not aligned (Figure 19).

There is a trade-off in “scrambling” the interconnections with either interwoven circuit board traces or through software. This is extremely cumbersome (if not impossible) to do with byte-oriented computer architectures. The 8051’s unique set of Boolean instructions makes it simple to move individual bits between arbitrary locations.

---

**4. BOOLEAN PROCESSING INSTRUCTIONS**

The commonly accepted terms for tasks at either end of the computational vs. control application spectrum are, respectively, “number-crunching” and “bit-banging”.

---

AFN-01502A-27

2-25
Example 26—Re-ordering I/O Port Configuration

```
OUT P2 RRC A MOVE ORIGINAL ACC 0 INTO CY
MOV P2 6-C STORE CARRY TO PIN P26
RRC A MOVE ORIGINAL ACC 1 INTO CY
MOV P2 5-C STORE CARRY TO PIN P25
RRC A MOVE ORIGINAL ACC 2 INTO CY
MOV P2 4-C STORE CARRY TO PIN P24
RRC A MOVE ORIGINAL ACC 3 INTO CY
MOV P2 3-C STORE CARRY TO PIN P23
RRC A MOVE ORIGINAL ACC 4 INTO CY
MOV P2 2-C STORE CARRY TO PIN P22
RET
```

Solving Combinatorial Logic Equations — ANL, ORL

Virtually all hardware designers are familiar with the problem of solving complex functions using combinatorial logic. The technologies involved may vary greatly, from multiple contact relay logic, vacuum tubes, TTL, or CMOS to more esoteric approaches like fluidics, but in each case the goal is the same: a Boolean (true false) function is computed on a number of Boolean variables.
Figure 20 shows the logic diagram for an arbitrary function of six variables named U through Z using standard logic and relay logic symbols. Each is a solution of the equation.

\[ Q = (U \cdot (V + W)) + (X \cdot \overline{Y}) + Z \]

(While this equation could be reduced using Karnaugh Maps or algebraic techniques, that is not the purpose of this example. Even a minor change to the function equation would require re-reducing from scratch.)

Most digital computers can solve equations of this type with standard word-wide logical instructions and conditional jumps. Still, such software solutions seem somewhat sloppy because of the many paths through the program the computation can take.

Assume U and V are input pins being read by different input ports, W and X are status bits for two peripheral controllers (read as I/O ports), and Y and Z are software flags set or cleared earlier in the program. The end result must be written to an output pin on some third port.

For the sake of comparison we will implement this function with software drawn from three proper subsets of the MCS-51™ instruction set. The first two implementations follow the flow chart shown in Figure 21. Program flow would embark on a route down a test-and-branch tree and leaves either the "True" or "Not True" exit ASAP. These exits then write the output port with the data previously written to the same port with the result bit respectively one or zero.

In the first case, we assume there are no instructions for addressing individual bits other than special flags like the carry. This is typical of many older microprocessors and mainframe computers designed for number-crunching. MCS-51™ mnemonics are used here, though for most other machines the issue would be even further clouded by their use of operation-specific mnemonics like INPUT, OUTPUT, ILOAD, STORE, etc., instead of the universal MOV.
Cumbersome, to say the least, and error prone. It would be hard to prove the above example worked in all cases without an exhaustive test.

Each move/mask/conditional jump instruction sequence may be replaced by a single bit-test instruction thanks to direct bit addressing. But the algorithm would be equally convoluted.

Example 28 — Software Solution to Logic Function of Figure 20, Using only Bit-Test Instructions

```
OUTBUF EQU 22H . OUTPUT PIN STATE MAP
TESTV  MOV A,#00000000B
ANL A,#00000001B
JZ TESTX
TESTU  MOV A,#00000000B
ANL A,#00000001B
JZ TESTY
TESTY MOV A,#00000000B
ANL A,#00000001B
JZ TESTZ
TESTZ MOV A,#00000000B
ANL A,#00000001B
JZ TEST0
CLR_D MOV A,OUTBUF
SETB_D MOV A,#11111111B
JMP

A more elegant and efficient 8051 implementation uses the Boolean ANL and ORL functions to generate the output function using straight-line code. These instructions perform the corresponding logical operations between the carry flag (“Boolean Accumulator”) and the addressed bit, leaving the result in the carry. Alternate forms of each instruction (specified in the assembly language by placing a slash before the bit name) use the complement of the bit's state as the input operand.
```

These instructions may be “strung together” to simulate a multiple input logic gate. When finished, the carry flag contains the result, which may be moved directly to the destination or output pin. No flow chart is needed — it is simple to code directly from the logic diagrams in Figure 20.

Example 29 — Software Solution to Logic Function of Figure 20, Using the MCS-51 (TM) Unique Logical Instructions on Boolean Variables

```
OUTBUF EQU 22H . OUTPUT PIN STATE MAP
TESTV  MOV A,#00000000B
ANL A,#00000001B
JZ TESTX
TESTU  MOV A,#00000000B
ANL A,#00000001B
JZ TESTY
TESTY MOV A,#00000000B
ANL A,#00000001B
JZ TESTZ
TESTZ MOV A,#00000000B
ANL A,#00000001B
JZ TEST0
CLR_D MOV A,OUTBUF
SETB_D MOV A,#11111111B
JMP

A more elegant and efficient 8051 implementation uses the Boolean ANL and ORL functions to generate the output function using straight-line code. These instructions perform the corresponding logical operations between the carry flag (“Boolean Accumulator”) and the addressed bit, leaving the result in the carry. Alternate forms of each instruction (specified in the assembly language by placing a slash before the bit name) use the complement of the bit's state as the input operand.
```

Simplicity itself. Fast, flexible, reliable, easy to design, and easy to debug.

The Boolean features are useful and unique enough to warrant a complete Application Note of their own. Additional uses and ideas are presented in Application Note AP-70, Using the Intel® MCS-51® Boolean Processing Capabilities, publication number 121519.

5. ON-CHIP PERIPHERAL FUNCTION OPERATION AND INTERFACING

I/O Ports

The 1/0 port versatility results from the "quasi-bidirectional" output structure depicted in Figure 22. This is effectively the structure of ports 1, 2, and 3 for normal 1/0 operations. On port 0 resistor R2 is disabled except during multiplexed bus operations, providing

![Figure 22. Pseudo-bidirectional I/O port circuitry](AFN-01502A-30)

2-28
essentially open-collector outputs. For full electrical characteristics see the User's Manual.)

An output latch bit associated with each pin is updated by direct addressing instructions when that port is the destination. The latch state is buffered to the outside world by R1 and Q1, which may drive a standard TTL input. (In TTL terms, Q1 and R1 resemble an open-collector output with a pull-up resistor to Vcc.)

R2 and Q2 represent an "active pull-up" device enabled momentarily when a 0 previously output changes to a 1. This "jerks" the output pin to a 1 level more quickly than the passive pull-up, improving rise-time significantly if the pin is driving a capacitive load. Note that the active pull-up is only activated on 0-to-1 transitions at the output latch (unlike the 8048, in which Q2 is activated whenever a 1 is written out).

Operations using an input port or pin as the source operand use the logic level of the pin itself, rather than the output latch contents. This level is affected by both the microcomputer itself and whatever device the pin is connected to externally. The value read is essentially the "OR-tied" function of Q1 and the external device. If the external device is high-impedance, such as a logic gate input or a three state output in the third state, then reading a pin will reflect the logic level previously output. To use a pin for input, the corresponding output latch must be set. The external device may then drive the pin with either a high or low logic signal. Thus the same port may be used as both input and output by writing ones to all pins used as outputs on input operations, and ignoring all pins used as output on an input operation.

In one operand instructions (INC, DEC, DJNZ and the Boolean CPI) the output latch rather than the input pin level is used as the source data. Similarly, two operand instructions using the port as both one source and the destination (ANL, ORL, XRL) use the output latches. This ensures that latch bits corresponding to pins used as inputs will not be cleared in the process of executing these instructions.

The Boolean operation JBC tests the output latch bit, rather than the input pin, in deciding whether or not to jump. Like the byte-wise logical operations, Boolean operations which modify individual pins of a port leave the other bits of the output latch unchanged.

A good example of how these modes may play together may be taken from the host-processor interface expected by an 8243 I/O expander. Even though the 8051 does not include 8048-type instructions for interfacing with an 8243, the parts can be interconnected (Figure 23) and the protocol may be emulated with simple software.

---

Example 30 — Mixing Parallel Output, Input, and Control Strobes on Port 2

---

Serial Port and Timer applications

Configuring the 8051's Serial Port for a given data rate and protocol requires essentially three short sections of software. On power-up or hardware reset the serial port and timer control words must be initialized to the appropriate values. Additional software is also needed in the transmit routine to load the serial port data register and in the receive routine to unload the data as it arrives.

This is best illustrated through an arbitrary example. Assume the 8051 will communicate with a CRT operating at 2400 baud (bits per second). Each character is transmitted as seven data bits, odd parity, and one stop bit. This results in a character rate of 2400 10=240 characters per second.

For the sake of clarity, the transmit and receive subroutines are driven by simple-minded software status polling code rather than interrupts. (It might help to refer back to Figures 7-9 showing the control word formats.) The serial port must be initialized to 8-bit UART mode (M0=M1=0). A flag indicating that the transmit register is free for more data will be artificially set in order to let the output software know the output register is available. This can all be set up with one instruction.

Example 31 — Serial Port Mode and Control Bits

---

Figure 23. Connecting an 8051 with an 8243 I/O Expander

---
Timer 1 will be used in auto-reload mode as a data rate generator. To achieve a data rate of 2400 baud, the timer must divide the 1 MHz internal clock by 32 x (desired data rate):

\[
\frac{1 \times 10^6}{(32) (2400)}
\]

which equals 13.02 rounded down to 13 instruction cycles. The timer must reload the value -13, or 0F3H. (ASM5I will accept both the signed decimal or hexadecimal representations.)

**Example 32 — Initializing Timer Mode and Control Bits**

```
; TINIT INITIALIZE TIMER 1 FOR
; AUTO-RELOAD AT 32*2400 Hz
; (TO USE AS GATED 16-BIT COUNTER )
TINIT MOV TOCN:10100100
MOV TH1<-13
SETB TR1
```

A simple subroutine to transmit the character passed to it in the accumulator must first compute the parity bit, insert it into the data byte, wait until the transmitter is available, output the character, and return. This is nearly as easy said as done.

**Example 33 — Code for UART Output, Adding Parity, Transmitter Loading**

```
; SP_OUT ADD ODD PARITY TO ACC AND
; TRANSMIT WHEN SERIAL PORT READY
SP_OUT MOV C,P
CPL E
MOV ACC.7,C
CLR TI
MOV SBUF.A
RET
```

A simple minded routine to wait until a character is received, set the carry flag if there is an odd-parity error, and return the masked seven-bit code in the accumulator is equally short.

**Example 34 — Code for UART Reception and Parity Verification**

```
; SP_IN INPUT NEXT CHARACTER FROM SERIAL PORT
; SET CARRY IF ODD-PARITY ERROR
SP_IN JNB RI, *;
CLR RI
MOV A,SBUF
CPL C
MOV C,P
CPL RI
MOV RI,A
MOV RI.7
RET
```

6. SUMMARY

This Application Note has described the architecture, instruction set, and on-chip peripheral features of the first three members of the MCS-51™ microcomputer family. The examples used throughout were admittedly (and necessarily) very simple. Additional examples and techniques may be found in the MCS-51™ User's Manual and other application notes written for the MCS-48™ and MCS-51™ families.

Since its introduction in 1977, the MCS-48™ family has become the industry standard single-chip microcomputer. The MCS-51™ architecture expands the addressing capabilities and instruction set of its predecessor while ensuring flexibility for the future, and maintaining basic software compatibility with the past.

Designers already familiar with the 8048 or 8049 will be able to take with them the education and experience gained from past designs as ever-increasing system performance demands force them to move on to state-of-the-art products. Newcomers will find the power and regularity of the 8051 instruction set an advantage in streamlining both the learning and design processes.

Microcomputer system designers will appreciate the 8051 as basically a single-chip solution to many problems which previously required board-level computers. Designers of real-time control systems will find the high execution speed, on-chip peripherals, and interrupt capabilities vital in meeting the timing constraints of products previously requiring discrete logic designs. And designers of industrial controllers will be able to convert ladder diagrams directly from tested-and-true TTL or relay-logic designs to microcomputer software, thanks to the unique Boolean processing capabilities.

It has not been the intent of this note to gloss over the difficulty of designing microcomputer-based systems. To be sure, the hardware and software design aspects of any new computer system are nontrivial tasks. However, the system speed and level of integration of the MCS-51™ microcomputers, the power and flexibility of the instruction set, and the sophisticated assembler and other support products combine to give both the hardware and software designer as much of a head start on the problem as possible.
Using the Intel MCS®-51 Boolean Processing Capabilities

JOHN WHARTON
MICROCONTROLLER APPLICATIONS
1.0 INTRODUCTION

The Intel microcontroller family now has three new members: the Intel® 8031, 8051, and 8751 single-chip microcomputers. These devices, shown in Figure 1, will allow whole new classes of products to benefit from recent advances in Integrated Electronics. Thanks to Intel's new HMOS technology, they provide larger program and data memory spaces, more flexible I/O and peripheral capabilities, greater speed, and lower system cost than any previous-generation single-chip microcomputer.

Table 1 summarizes the quantitative differences between the members of the MCS®-48 and 8051 families. The 8751 contains 4K bytes of EPROM program memory fabricated on-chip, while the 8051 replaces the EPROM with 4K bytes of lower-cost mask-programmed ROM. The 8031 has no program memory on-chip; instead, it accesses up to 64K bytes of program memory from external memory. Otherwise, the three new family members are identical. Throughout this Note, the term “8051” will represent all members of the 8051 Family, unless specifically stated otherwise.

The CPU in each microcomputer is one of the industry's fastest and most efficient for numerical calculations on byte operands. But controllers often deal with bits, not bytes: in the real world, switch contacts can only be open or closed, indicators should be either lit or dark, motors are either turned on or off, and so forth. For such control situations the most significant aspect of the MCS®-51 architecture is its complete hardware support for one-bit, or Boolean variables (named in honor of Mathematician George Boole) as a separate data type.

The 8051 incorporates a number of special features which support the direct manipulation and testing of individual bits and allow the use of single-bit variables in performing logical operations. Taken together, these features are referred to as the MCS-51 Boolean Processor. While the bit-processing capabilities alone would be adequate to solve many control applications, their true power comes when they are used in conjunction with the microcomputer's byte-processing and numerical capabilities.

Many concepts embodied by the Boolean Processor will certainly be new even to experienced microcomputer system designers. The purpose of this Application Note is to explain these concepts and show how they are used.

For detailed information on these parts refer to the Intel Microcontroller Handbook, order number 210918. The instruction set, assembly language, and use of the 8051 assembler (ASM51) are further described in the MCS®-51 Macro Assembler User's Guide for DOS Systems, order number 122753.

<table>
<thead>
<tr>
<th>EPROM Program Memory</th>
<th>ROM Program Memory</th>
<th>External Program Memory</th>
<th>Program Memory (Int/Max)</th>
<th>Data Memory (Bytes)</th>
<th>Inst. Cycle Time</th>
<th>Input/Output Pins</th>
<th>Interrupt Sources</th>
<th>Reg. Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>8748</td>
<td>8048</td>
<td>8035</td>
<td>1K 4K</td>
<td>64</td>
<td>2.5 μs</td>
<td>27</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>—</td>
<td>8049</td>
<td>8039</td>
<td>2K 4K</td>
<td>128</td>
<td>1.36 μs</td>
<td>27</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8751</td>
<td>8051</td>
<td>8031</td>
<td>4K 64K</td>
<td>128</td>
<td>1.0 μs</td>
<td>32</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 1. 8051 Family Pinout Diagram
2.0 BOOLEAN PROCESSOR OPERATION

The Boolean Processing capabilities of the 8051 are based on concepts which have been around for some time. Digital computer systems of widely varying designs all have four functional elements in common (Figure 2):

- a central processor (CPU) with the control, timing, and logic circuits needed to execute stored instructions;
- a memory to store the sequence of instructions making up a program or algorithm;
- data memory to store variables used by the program;
- and
- some means of communicating with the outside world.

The CPU usually includes one or more accumulators or special registers for computing or storing values during program execution. The instruction set of such a processor generally includes, at a minimum, operation classes to perform arithmetic or logical functions on program variables, move variables from one place to another, cause program execution to jump or conditionally branch based on register or variable states, and instructions to call and return from subroutines. The program and data memory functions sometimes share a single memory space, but this is not always the case. When the address spaces are separated, program and data memory need not even have the same basic word width.

A digital computer’s flexibility comes in part from combining simple fast operations to produce more complex (albeit slower) ones, which in turn link together eventually solving the problem at hand. A four-bit CPU executing multiple precision subroutines can, for example, perform 64-bit addition and subtraction. The subroutines could in turn be building blocks for floating-point multiplication and division routines. Eventually, the four-bit CPU can simulate a far more complex “virtual” machine.

In fact, any digital computer with the above four functional elements can (given time) complete any algorithm (though the proverbial room full of chimpanzees at word processors might first re-create Shakespeare’s classics and this Application Note)! This fact offers little consolation to product designers who want programs to run as quickly as possible. By definition, a real-time control algorithm must proceed quickly enough to meet the preordained speed constraints of other equipment.

One of the factors determining how long it will take a microcomputer to complete a given chore is the number of instructions it must execute. What makes a given computer architecture particularly well- or poorly-suited for a class of problems is how well its instruction set matches the tasks to be performed. The better the “primitive” operations correspond to the steps taken by the control algorithm, the lower the number of instructions needed, and the quicker the program will run. All else being equal, a CPU supporting 64-bit arithmetic directly could clearly perform floating-point math faster than a machine bogged-down by multiple-precision subroutines. In the same way, direct support for bit manipulation naturally leads to more efficient programs handling the binary input and output conditions inherent in digital control problems.
Processing Elements

The introduction stated that the 8051's bit-handling capabilities alone would be sufficient to solve some control applications. Let's see how the four basic elements of a digital computer—a CPU with associated registers, program memory, addressable data RAM, and I/O capability—relate to Boolean variables.

CPU. The 8051 CPU incorporates special logic devoted to executing several bit-wide operations. All told, there are 17 such instructions, all listed in Table 2. Not shown are 94 other (mostly byte-oriented) 8051 instructions.

Program Memory. Bit-processing instructions are fetched from the same program memory as other arithmetic and logical operations. In addition to the instructions of Table 2, several sophisticated program control features like multiple addressing modes, subroutine nesting, and a two-level interrupt structure are useful in structuring Boolean Processor-based programs.

Boolean instructions are one, two, or three bytes long, depending on what function they perform. Those involving only the carry flag have either a single-byte opcode or an opcode followed by a conditional-branch destination byte (Figure 3a). The more general instructions add a "direct address" byte after the opcode to specify the bit affected, yielding two or three byte encodings (Figure 3b). Though this format allows potentially 256 directly addressable bit locations, not all of them are implemented in the 8051 family.

Table 2. MCS-51™ Boolean Processing Instruction Subset

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETB C</td>
<td>Set Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SETB bit</td>
<td>Set direct Bit</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CLR C</td>
<td>Clear Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CLR bit</td>
<td>Clear direct bit</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CPL C</td>
<td>Complement Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL bit</td>
<td>Complement direct bit</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MOV C.bit</td>
<td>Move direct bit to Carry flag</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MOV bit.C</td>
<td>Move Carry flag to direct bit</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ANL C.bit</td>
<td>AND direct bit to Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ANL C.bit</td>
<td>AND complement of direct bit to Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL C.bit</td>
<td>OR direct bit to Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL C.bit</td>
<td>OR complement of direct bit to Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JC rel</td>
<td>Jump if Carry is flag is set</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JNC rel</td>
<td>Jump if No Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JB bit.rel</td>
<td>Jump if direct Bit set</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>JNB bit.rel</td>
<td>Jump if direct Bit Not set</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>JBC bit.rel</td>
<td>Jump if direct Bit is set &amp; Clear bit</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Address mode abbreviations

C—Carry flag.
bit—128 software flags, any I/O pin, control or status bit.
rel—All conditional jumps include an 8-bit offset byte. Range is +127 −128 bytes relative to first byte of the following instruction.

All mnemonics copyrighted © Intel Corporation 1980.

Figure 3. Bit Addressing Instruction Formats

a.) Carry Control and Test Instructions

b.) Bit Manipulation and Test Instructions
### RAM Bit Addresses

<table>
<thead>
<tr>
<th>RAM Byte (MSB)</th>
<th>(LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FH ~ 2FH</td>
<td></td>
</tr>
<tr>
<td>6F</td>
<td>6E</td>
</tr>
<tr>
<td>6D</td>
<td>6C</td>
</tr>
<tr>
<td>6B</td>
<td>6A</td>
</tr>
<tr>
<td>69</td>
<td>68</td>
</tr>
<tr>
<td>2FH</td>
<td>7E</td>
</tr>
<tr>
<td>7D</td>
<td>7C</td>
</tr>
<tr>
<td>7B</td>
<td>7A</td>
</tr>
<tr>
<td>79</td>
<td>78</td>
</tr>
<tr>
<td>2EH</td>
<td>77</td>
</tr>
<tr>
<td>75</td>
<td>74</td>
</tr>
<tr>
<td>73</td>
<td>72</td>
</tr>
<tr>
<td>71</td>
<td>70</td>
</tr>
<tr>
<td>2DH</td>
<td>67</td>
</tr>
<tr>
<td>66</td>
<td>65</td>
</tr>
<tr>
<td>64</td>
<td>63</td>
</tr>
<tr>
<td>62</td>
<td>61</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>2CH</td>
<td>5F</td>
</tr>
<tr>
<td>5E</td>
<td>5D</td>
</tr>
<tr>
<td>5C</td>
<td>5B</td>
</tr>
<tr>
<td>5A</td>
<td>59</td>
</tr>
<tr>
<td>58</td>
<td></td>
</tr>
<tr>
<td>2AH</td>
<td>57</td>
</tr>
<tr>
<td>56</td>
<td>55</td>
</tr>
<tr>
<td>54</td>
<td>53</td>
</tr>
<tr>
<td>52</td>
<td>51</td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>29H</td>
<td>4F</td>
</tr>
<tr>
<td>4E</td>
<td>4D</td>
</tr>
<tr>
<td>4C</td>
<td>4B</td>
</tr>
<tr>
<td>4A</td>
<td>49</td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>28H</td>
<td>47</td>
</tr>
<tr>
<td>46</td>
<td>45</td>
</tr>
<tr>
<td>44</td>
<td>43</td>
</tr>
<tr>
<td>42</td>
<td>41</td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>27H</td>
<td>3F</td>
</tr>
<tr>
<td>3E</td>
<td>3D</td>
</tr>
<tr>
<td>3C</td>
<td>3B</td>
</tr>
<tr>
<td>3A</td>
<td>39</td>
</tr>
<tr>
<td>38</td>
<td></td>
</tr>
<tr>
<td>26H</td>
<td>37</td>
</tr>
<tr>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
</tr>
<tr>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>25H</td>
<td>2F</td>
</tr>
<tr>
<td>2E</td>
<td>2D</td>
</tr>
<tr>
<td>2C</td>
<td>2B</td>
</tr>
<tr>
<td>2A</td>
<td>29</td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>24H</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>24</td>
<td>23</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>23H</td>
<td>1F</td>
</tr>
<tr>
<td>1E</td>
<td>1D</td>
</tr>
<tr>
<td>1C</td>
<td>1B</td>
</tr>
<tr>
<td>1A</td>
<td>19</td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>22H</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>21H</td>
<td>0F</td>
</tr>
<tr>
<td>0E</td>
<td>0D</td>
</tr>
<tr>
<td>0C</td>
<td>0B</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
</tr>
<tr>
<td>08</td>
<td></td>
</tr>
<tr>
<td>20H</td>
<td>07</td>
</tr>
<tr>
<td>06</td>
<td>05</td>
</tr>
<tr>
<td>04</td>
<td>03</td>
</tr>
<tr>
<td>02</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

### Direct Byte Address (MSB) | Bit Addresses (LSB) | Hardware Register Symbol
---|-----------------------|---------------------
0FFH | F7 F6 F5 F4 F3 F2 F1 F0 | B
0F0H | E7 E6 E5 E4 E3 E2 E1 E0 | ACC
0E0H | D7 D6 D5 D4 D3 D2 D1 D0 | PSW
0D0H | C7 C6 C5 C4 C3 C2 C1 C0 | IP
0C0H | B7 B6 B5 B4 B3 B2 B1 B0 | P3
0A0H | A7 A6 A5 A4 A3 A2 A1 A0 | P2
98H  | 9F 9E 9D 9C 9B 9A 99 98  | SCON
90H  | 97 96 95 94 93 92 91 90  | P1
88H  | 8F 8E 8D 8C 8B 8A 89 88  | TCON
80H  | 87 86 85 84 83 82 81 80  | P0

### Figure 4. Bit Address Maps

**Data Memory.** The instructions in Figure 3b can operate directly upon 144 general purpose bits forming the Boolean processor "RAM." These bits can be used as software flags or to store program variables. Two operand instructions use the CPU's carry flag ("C") as a special one-bit register: in a sense, the carry is a "Boolean accumulator" for logical operations and data transfers.

**Input/Output.** All 32 I/O pins can be addressed as individual inputs, outputs, or both, in any combination. Any pin can be a control strobe output, status (Test) input, or serial I/O link implemented via software. An additional 33 individually addressable bits reconfigure, control, and monitor the status of the CPU and all on-chip peripheral functions (timer counters, serial port modes, interrupt logic, and so forth).
### Symbol Position Name and Significance

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CY</td>
<td>PSW.7</td>
<td>Carry flag. Set/cleared by hardware or software during certain arithmetic and logical instructions.</td>
</tr>
<tr>
<td>AC</td>
<td>PSW.6</td>
<td>Auxiliary Carry flag. Set/cleared by hardware during addition or subtraction instructions to indicate carry or borrow out of bit 3.</td>
</tr>
<tr>
<td>F0</td>
<td>PSW.5</td>
<td>Flag 0. Set/cleared/tested by software as a user-defined status flag.</td>
</tr>
<tr>
<td>RS1</td>
<td>PSW.4</td>
<td>Register bank Select control bits.</td>
</tr>
<tr>
<td>RS0</td>
<td>PSW.3</td>
<td>1 &amp; 0. Set/cleared by software to determine working register bank (see Note).</td>
</tr>
</tbody>
</table>

#### Note

The contents of (RS1, RS0) enable the working register banks as follows:

- (0,0) - Bank 0 (00H–07H)
- (0,1) - Bank 1 (08H–0FH)
- (1,0) - Bank 2 (10H–17H)
- (1,1) - Bank 3 (18H–1FH)

### Direct Bit Addressing

The most significant bit of the direct address byte selects one of two groups of bits. Values between 0 and 127 (00H and 7FH) define bits in a block of 32 bytes of on-chip RAM, between RAM addresses 20H and 2FH (Figure 4a). They are numbered consecutively from the lowest-order byte's lowest-order bit through the highest-order byte's highest-order bit.
Notice the column labeled “Symbol” in Figure 5. Bits with special meanings in the PSW and other registers have corresponding symbolic names. General-purpose (as opposed to carry-specific) instructions may access the carry like any other bit by using the mnemonic CY in place of C, P0, P1, P2, and P3 are the 8051’s four I/O ports: secondary functions assigned to each of the eight pins of P3 are shown in Figure 6.

Figure 7 shows the last four bit addressable registers. TCON (Timer Control) and SCON (Serial port Control) control and monitor the corresponding peripherals, while IE (Interrupt Enable) and IP (Interrupt Priority) enable and prioritize the five hardware interrupt sources. Like the reserved hardware register addresses, the five bits not implemented in IE and IP should not be accessed: they can not be used as software flags.

Addressable Register Set. There are 20 special function registers in the 8051, but the advantages of bit addressing only relate to the 11 described below. Five potentially bit-addressable register addresses (0C0H, 0C8H, 0D8H, 0E8H, & 0F8H) are being reserved for possible future expansion in microcomputers based on the MCS-51 architecture. Reading or writing non-existent registers in the 8051 series is pointless, and may cause unpredictable results. Byte-wide logical operations can be used to manipulate bits in all non-bit addressable registers and RAM.
### Symbol Position Name and Significance

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
</table>
| TF1    | TCON.7   | Timer 1 overflow Flag.  
|        |          | Set by hardware on timer/counter overflow. Cleared when interrupt processed. |
| TR1    | TCON.6   | Timer 1 Run control bit.  
|        |          | Set/cleared by software to turn timer/counter on/off. |
| TF0    | TCON.5   | Timer 0 overflow Flag.  
|        |          | Set by hardware on timer/counter overflow. Cleared when interrupt processed. |
| TR0    | TCON.4   | Timer 0 Run control bit.  
|        |          | Set/cleared by software to turn timer/counter on/off. |

#### a.) TCON—Timer/Counter Control/Status Register

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
</table>
| SM0    | SCON.7   | Serial port Mode control bit 0.  
|        |          | Set/cleared by software (see note). |
| SM1    | SCON.6   | Serial port Mode control bit 1.  
|        |          | Set/cleared by software (see note). |
| SM2    | SCON.5   | Serial port Mode control bit 2.  
|        |          | Set by software to disable reception of frames for which bit 8 is zero. |
| REN    | SCON.4   | Receiver Enable control bit.  
|        |          | Set/cleared by software to enable/disable serial data reception. |
| TB8    | SCON.3   | Transmit Bit 8.  
|        |          | Set/cleared by hardware to determine state of ninth data bit transmitted in 9-bit UART mode. |

#### b.) SCON—Serial Port Control/Status Register

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
</table>
| RB8    | SCON.2   | Receive Bit 8.  
|        |          | Set/cleared by hardware to indicate state of ninth data bit received. |
| TI     | SCON.1   | Transmit Interrupt flag.  
|        |          | Set by hardware when byte transmitted. Cleared by software after servicing. |
| RI     | SCON.0   | Receive Interrupt flag.  
|        |          | Set by hardware when byte received. Cleared by software after servicing. |

**Note:** the state of (SM0, SM1) selects:

- (0,0)—Shift register I/O expansion.
- (0,1)—8-bit UART, variable data rate.
- (1,0)—9-bit UART, fixed data rate.
- (1,1)—9-bit UART, variable data rate.

---

Figure 7. Peripheral Configuration Registers
The accumulator and B registers (A and B) are normally involved in byte-wide arithmetic, but their individual bits can also be used as 16 general software flags. Added with the 128 flags in RAM, this gives 144 general purpose variables for bit-intensive programs. The program status word (PSW) in Figure 5 is a collection of flags and machine status bits including the carry flag itself. Byte operations acting on the PSW can therefore affect the carry.

**Instruction Set**

Having looked at the bit variables available to the Boolean Processor, we will now look at the four classes of instructions that manipulate these bits. It may be helpful to refer back to Table 2 while reading this section.

**State Control.** Addressable bits or flags may be set, cleared, or logically complemented in one instruction cycle with the two-byte instructions SETB, CLR, and CPL. (The “B” affixed to SETB distinguishes it from the assembler “SET” directive used for symbol definition.) SETB and CLR are analogous to loading a bit with a constant: 1 or 0. Single byte versions perform the same three operations on the carry.

The MCS-51 assembly language specifies a bit address in any of three ways:
- by a number or expression corresponding to the direct bit address (0–255):
• by the name or address of the register containing the bit, the dot operator symbol (a period: "."), and the bit’s position in the register (7-0):
• in the case of control and status registers, by the predefined assembler symbols listed in the first columns of Figures 5-7.

Bits may also be given user-defined names with the assembler "BIT" directive and any of the above techniques. For example, bit 5 of the PSW may be cleared by any of the four instructions.

USR_FLG BIT PSW.5 ; User Symbol Definition
CLR OD6H ; Absolute Addressing
CLR PSW.5 ; Use of Dot Operator
CLR FO ; Pre-Defined Assembler Symbol
CLR USR_FLG ; User-Defined Symbol

Data Transfers. The two-byte MOV instructions can transport any addressable bit to the carry in one cycle, or copy the carry to the bit in two cycles. A bit can be moved between two arbitrary locations via the carry by combining the two instructions. (If necessary, push and pop the PSW to preserve the previous contents of the carry.) These instructions can replace the multi-instruction sequence of Figure 8, a program structure appearing in controller applications whenever flags or outputs are conditionally switched on or off.

Logical Operations. Four instructions perform the logical-AND and logical-OR operations between the carry and another bit, and leave the results in the carry. The instruction mnemonics are ANL and ORL; the absence or presence of a slash mark ("/") before the source operand indicates whether to use the positive-logic value or the logical complement of the addressed bit. (The source operand itself is never affected.)

Bit-test Instructions. The conditional jump instructions "JC rei" (Jump on Carry) and "JNC rei" (Jump on Not Carry) test the state of the carry flag, branching if it is a one or zero, respectively. (The letters "rei" denote relative code addressing.) The three-byte instructions "JB bit.rei" and "JNB bit.rei" (Jump on Bit and Jump on Not Bit) test the state of any addressable bit in a similar manner. A fifth instruction combines the Jump on Bit and Clear operations. "JBC bit.rei" conditionally branches to the indicated address, then clears the bit in the same two cycle instruction. This operation is the same as the MCS-48 "JTF" instructions.

All 8051 conditional jump instructions use program counter-relative addressing, and all execute in two cycles. The last instruction byte encodes a signed displacement ranging from -128 to +127. During execution, the CPU adds this value to the incremented program counter to produce the jump destination. Put another way, a conditional jump to the immediately following instruction would encode 00H in the offset byte.

A section of program or subroutine written using only relative jumps to nearby addresses will have the same machine code independent of the code’s location. An assembled routine may be repositioned anywhere in memory, even crossing memory page boundaries, without having to modify the program or recompute destination addresses. To facilitate this flexibility, there is an unconditional "Short Jump" (SJMP) which uses relative addressing as well. Since a programmer would have quite a chore trying to compute relative offset values from one instruction to another, ASM51 automatically computes the displacement needed given only the destination address or label. An error message will alert the programmer if the destination is "out of range."

The so-called "Bit Test" instructions implemented on many other microprocessors simply perform the logical-AND operation between a byte variable and a constant mask, and set or clear a zero flag depending on the result. This is essentially equivalent to the 8051 "MOV C.bit" instruction. A second instruction is then needed to conditionally branch based on the state of the zero flag. This does not constitute abstract bit-addressing in the MCS-51 sense. A flag exists only as a field.
within a register: to reference a bit the programmer must know and specify both the encompassing register and the bit's position therein. This constraint severely limits the flexibility of symbolic bit addressing and reduces the machine's code-efficiency and speed.

**Interaction with Other Instructions.** The carry flag is also affected by the instructions listed in Table 3. It can be rotated through the accumulator, and altered as a side effect of arithmetic instructions. Refer to the User's Manual for details on how these instructions operate.

**Simple Instruction Combinations**

By combining general purpose bit operations with certain addressable bits, one can "custom build" several hundred useful instructions. All eight bits of the PSW can be tested directly with conditional jump instructions to monitor (among other things) parity and overflow status. Programmers can take advantage of 128 software flags to keep track of operating modes, resource usage, and so forth.

The Boolean instructions are also the most efficient way to control or reconfigure peripheral and I/O registers. All 32 I/O lines become "test pins," for example, tested by conditional jump instructions. Any output pin can be toggled (complemented) in a single instruction cycle. Setting or clearing the Timer Run flags (TR0 and TR1) turn the timer/counters on or off; polling the same flags elsewhere lets the program determine if a timer is running. The respective overflow flags (TF0 and TF1) can be tested to determine when the desired period or count has elapsed, then cleared in preparation for the next repetition. (For the record, these bits are all part of the TCON register, Figure 7a. Thanks to symbolic bit addressing, the programmer only needs to remember the mnemonic associated with each function. In other words, don't bother memorizing control word layouts.)

In the MCS-48 family, instructions corresponding to some of the above functions require specific opcodes. Ten different opcodes serve to clear complement the software flags F0 and F1, enable/disable each interrupt, and start/stop the timer. In the 8051 instruction set, just three opcodes (SETB, CLR, CPL) with a direct bit address appended perform the same functions. Two test instructions (JB and JNB) can be combined with bit addresses to test the software flags, the 8048 I/O pins T0, T1, and INT, and the eight accumulator bits, replacing 15 more different instructions.

Table 4a shows how 8051 programs implement software flag and machine control functions associated with special opcodes in the 8048. In every case the MCS-51 solution requires the same number of machine cycles, and executes 2.5 times faster.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A,Rn</td>
<td>Add register to Accumulator</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD A,direct</td>
<td>Add direct byte to Accumulator</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ADD A,@Ri</td>
<td>Add indirect RAM to Accumulator</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD A,#data</td>
<td>Add immediate data to Accumulator</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,Rn</td>
<td>Add register to Accumulator with Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,direct</td>
<td>Add direct byte to Accumulator with Carry flag</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,@Ri</td>
<td>Add indirect RAM to Accumulator with Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,#data</td>
<td>Add immediate data to Accumulator with Carry flag</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,Rn</td>
<td>Subtract register from Accumulator with borrow</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,direct</td>
<td>Subtract direct byte from Accumulator with borrow</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,@Ri</td>
<td>Subtract indirect RAM from Accumulator with borrow</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,#data</td>
<td>Subtract immediate data from Accumulator with borrow</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MUL AB</td>
<td>Multiply A &amp; B</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>DIV AB</td>
<td>Divide A by B</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>DA A</td>
<td>Decimal Adjust</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Accumulator Left through the Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RRC A</td>
<td>Rotate Accumulator Right through Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CJNE A,direct,rel</td>
<td>Compare direct byte to Acc &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CJNE A,#data,rel</td>
<td>Compare immediate to Acc &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CJNE Rn,#data,rel</td>
<td>Compare immed to register &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CJNE @Ri,#data,rel</td>
<td>Compare immed to indirect &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1980.
### Table 4a. Contrasting 8048 and 8051 Bit Control and Testing Instructions

<table>
<thead>
<tr>
<th>8048 Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
<th>μSec</th>
<th>8x51 Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
<th>μSec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR C</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>CLR C</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CPL F0</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>CPL F0</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Flag Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNC offset</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JNC rel</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>JF0 offset</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JB F0.rel</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>JB7 offset</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JB ACC.7.rel</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Peripheral Polling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JT0 offset</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JB T0.rel</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>JN1 offset</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JNB INTO.rel</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>JTF offset</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JBC TF0.rel</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Machine and Peripheral Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>START T</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>SETB TR0</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>EN</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>SETB EX0</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DIS TCNT1</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>CLR ET0</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4b. Replacing 8048 Instruction Sequences with Single 8x51 Instructions

<table>
<thead>
<tr>
<th>8048 Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
<th>μSec</th>
<th>8051 Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
<th>μSec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set carry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR C</td>
<td>= 2</td>
<td>2</td>
<td>5.0</td>
<td>SETB C</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CPL F0</td>
<td>= 2</td>
<td>2</td>
<td>5.0</td>
<td>SETB F0</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Set Software Flag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR F0</td>
<td>= 4</td>
<td>6</td>
<td>15.0</td>
<td>CPL P1.2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Complement Output Pin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL P1.#0FBH</td>
<td>= 2</td>
<td>2</td>
<td>5.0</td>
<td>CLR P1.2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Turn Off Output Pin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN A,P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XRL A,#04H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTL P1,A</td>
<td>= 4</td>
<td>6</td>
<td>15.0</td>
<td>CPL P1.2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Clear Flag in RAM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV R0,#FLGADR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV A,@R0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL A,#FLGMASK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV @R0,A</td>
<td>= 6</td>
<td>6</td>
<td>15.0</td>
<td>CLR USER_.FLG</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Table 4b. Replacing 8048 Instruction Sequences with Single 8x51 Instructions (Continued)

<table>
<thead>
<tr>
<th>8048 Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
<th>μSec</th>
<th>8x51 Instruction</th>
<th>Bytes</th>
<th>Cycles &amp; μ Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flag Testing:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump if Software Flag is 0</td>
<td></td>
<td></td>
<td></td>
<td>JNB F0.rel</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>JF0 $+4</td>
<td>JMP offset</td>
<td>= 4</td>
<td>4</td>
<td>10.0</td>
<td>JNB ACC.7.rel</td>
<td>3</td>
</tr>
<tr>
<td>Jump if Accumulator bit is 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPL A</td>
<td>JB7 offset</td>
<td>= 4</td>
<td>4</td>
<td>10.0</td>
<td>JNB ACC.7.rel</td>
<td>3</td>
</tr>
<tr>
<td>Peripheral Polling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test if Input Pin is Grounded</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN A,P1</td>
<td>CPL A</td>
<td>JB3 offset</td>
<td>= 4</td>
<td>5</td>
<td>12.5</td>
<td>JNB P1.3.rel</td>
</tr>
<tr>
<td>Test if Interrupt Pin is High</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JN1 $+4</td>
<td>JMP offset</td>
<td>= 4</td>
<td>4</td>
<td>10.0</td>
<td>JB INTO.rel</td>
<td>3</td>
</tr>
</tbody>
</table>

3.0 BOOLEAN PROCESSOR APPLICATIONS

So what? Then what does all this buy you?

**Qualitatively**, nothing. All the same capabilities could be (and often have been) implemented on other machines using awkward sequences of other basic operations. As mentioned earlier, any CPU can solve any problem given enough time.

**Quantitatively**, the differences between a solution allowed by the 8051 and those required by previous architectures are numerous. What the 8051 Family buys you is a faster, cleaner, lower-cost solution to microcontroller applications.

The opcode space freed by condensing many specific 8048 instructions into a few general operations has been used to add new functionality to the MCS-51 architecture—both for byte and bit operations. 144 software flags replace the 8048's two. These flags (and the carry) may be directly set, not just cleared and complemented, and all can be tested for either state, not just one. Operating mode bits previously inaccessible may be read, tested, or saved. Situations where the 8051 instruction set provides new capabilities are contrasted with 8048 instruction sequences in Table 4b. Here the 8051 speed advantage ranges from 5x to 15x!

Combining Boolean and byte-wide instructions can produce great synergy. An MCS-51 based application will prove to be:

- simpler to write since the architecture correlates more closely with the problems being solved;
- easier to debug because more individual instructions have no unexpected or undesirable side-effects;
- more byte efficient due to direct bit addressing and program counter relative branching;
- faster running because fewer bytes of instruction need to be fetched and fewer conditional jumps are processed;
- lower cost because of the high level of system-integration within one component.

These rather unabashed claims of excellence shall not go unsubstantiated. The rest of this chapter examines less trivial tasks simplified by the Boolean processor. The first three compare the 8051 with other microprocessors; the last two go into 8051-based system designs in much greater depth.

**Design Example #1—Bit Permutation**

First off, we'll use the bit-transfer instructions to permute a lengthy pattern of bits.
A steadily increasing number of data communication products use encoding methods to protect the security of sensitive information. By law, interstate financial transactions involving the Federal banking system must be transmitted using the Federal Information Processing Data Encryption Standard (DES).

Basically, the DES combines eight bytes of "plaintext" data (in binary, ASCII, or any other format) with a 56-bit "key", producing a 64-bit encrypted value for transmission. At the receiving end the same algorithm is applied to the incoming data using the same key, reproducing the original eight byte message. The algorithm used for these permutations is fixed; different user-defined keys ensure data privacy.

It is not the purpose of this note to describe the DES in any detail. Suffice it to say that encryption/decryption is a long, iterative process consisting of rotations, exclusive -OR operations, function table look-ups, and an extensive (and quite bizarre) sequence of bit permutation, packing, and unpacking steps. (For further details refer to the June 21, 1979 issue of Electronics magazine.) The bit manipulation steps are included, it is rumored, to impede a general purpose digital supercomputer trying to "break" the code. Any algorithm implementing the DES with previous generation microprocessors would spend virtually all of its time diddling bits.

The bit manipulation performed is typified by the Key Schedule Calculation represented in Figure 9. This step is repeated 16 times for each key used in the course of a transmission. In essence, a seven-byte, 56-bit "Shifted Key Buffer" is transformed into an eight-byte, "Permutation Buffer" without altering the shifted Key. The arrows in Figure 9 indicate a few of the translation steps. Only six bits of each byte of the Permutation Buffer are used; the two high-order bits of each byte are cleared. This means only 48 of the 56 Shifted Key Buffer bits are used in any one iteration.

Different microprocessor architectures would best implement this type of permutation in different ways. Most approaches would share the steps of Figure 10a:
- Initialize the Permutation Buffer to default state (ones or zeroes);
- Isolate the state of a bit of a byte from the Key Buffer. Depending on the CPU, this might be accomplished by rotating a word of the Key Buffer through a carry flag or testing a bit in memory or an accumulator against a mask byte;
- Perform a conditional jump based on the carry or zero flag if the Permutation Buffer default state is correct;
- Otherwise reverse the corresponding bit in the permutation buffer with logical operations and mask bytes.

Each step above may require several instructions. The last three steps must be repeated for all 48 bits. Most microprocessors would spend 300 to 3,000 microseconds on each of the 16 iterations.

Notice, though, that this flow chart looks a lot like Figure 8. The Boolean Processor can permute bits by simply moving them from the source to the carry to the destination—a total of two instructions taking four bytes and three microseconds per bit. Assume the Shifted Key Buffer and Permutation Buffer both reside in bit-addressable RAM, with the bits of the former assigned symbolic names SKB_1, SKB_2, ..., SKB_56, and that the bytes of the latter are named PB_1, ..., PB_8. Then working from Figure 9, the software for the permutation algorithm would be that of Example 1a. The total routine length would be 192 bytes, requiring 144 microseconds.

![Figure 9. DES Key Schedule Transformation](image-url)
Figure 10a. Flowchart for Key Permutation Attempted with a Byte Processor
Figure 10b. DES Key Permutation with Boolean Processor
The algorithm of Figure 10b is just slightly more efficient in this time-critical application and illustrates the synergy of an integrated byte and bit processor. The bits needed for each byte of the Permutation Buffer are assimilated by loading each bit into the carry (1 μs.) and shifting it into the accumulator (1 μs.). Each byte is stored in RAM when completed. Forty-eight bits thus need a total of 112 instructions, some of which are listed in Example 1b.

Worst-case execution time would be 112 microseconds, since each instruction takes a single cycle. Routine length would also decrease, to 168 bytes. (Actually, in the context of the complete encryption algorithm, each permuted byte would be processed as soon as it is assimilated—saving memory and cutting execution time by another 8 μs.)

To date, most banking terminals and other systems using the DES have needed special boards or peripheral controller chips just for the encryption/decryption process, and still more hardware to form a serial bit stream for transmission (Figure 11a). An 8051 solution could pack most of the entire system onto the one chip (Figure 11b). The whole DES algorithm would require less than one-fourth of the on-chip program memory, with the remaining bytes free for operating the banking terminal (or whatever) itself.

Moreover, since transmission and reception of data is performed through the on-board UART, the unencrypted data (plaintext) never even exists outside the microcomputer! Naturally, this would afford a high degree of security from data interception.

---

**Example 1. DES Key Permutation Software.**

a.) “Brute Force” technique

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV C,SKB_1</td>
<td>Store SKB_1 in C</td>
</tr>
<tr>
<td>MOV PB_1.1,C</td>
<td>Store PB_1.1 in C</td>
</tr>
<tr>
<td>MOV C,SKB_2</td>
<td>Store SKB_2 in C</td>
</tr>
<tr>
<td>MOV PB_4.0,C</td>
<td>Store PB_4.0 in C</td>
</tr>
<tr>
<td>MOV C,SKB_3</td>
<td>Store SKB_3 in C</td>
</tr>
<tr>
<td>MOV PB_2.5,C</td>
<td>Store PB_2.5 in C</td>
</tr>
<tr>
<td>MOV C,SKB_4</td>
<td>Store SKB_4 in C</td>
</tr>
<tr>
<td>MOV PB_1.0,C</td>
<td>Store PB_1.0 in C</td>
</tr>
<tr>
<td>...</td>
<td>Repeat 11 times</td>
</tr>
<tr>
<td>MOV C,SKB_55</td>
<td>Store SKB_55 in C</td>
</tr>
<tr>
<td>MOV PB_5.0,C</td>
<td>Store PB_5.0 in C</td>
</tr>
<tr>
<td>MOV C,SKB_56</td>
<td>Store SKB_56 in C</td>
</tr>
<tr>
<td>MOV PB_7.2,C</td>
<td>Store PB_7.2 in C</td>
</tr>
</tbody>
</table>

b.) Using Accumulator to Collect Bits

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR A</td>
<td>Clear Accumulator</td>
</tr>
<tr>
<td>MOV C,SKB_14</td>
<td>Store SKB_14 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV C,SKB_17</td>
<td>Store SKB_17 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV C,SKB_11</td>
<td>Store SKB_11 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV C,SKB_24</td>
<td>Store SKB_24 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV C,SKB_1</td>
<td>Store SKB_1 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV C,SKB_5</td>
<td>Store SKB_5 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV PB_L,A</td>
<td>Store PB_L in Accumulator</td>
</tr>
<tr>
<td>...</td>
<td>Repeat 11 times</td>
</tr>
<tr>
<td>MOV C,SKB_29</td>
<td>Store SKB_29 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV C,SKB_32</td>
<td>Store SKB_32 in C</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Left Once</td>
</tr>
<tr>
<td>MOV PB_8,A</td>
<td>Store PB_8 in Accumulator</td>
</tr>
</tbody>
</table>
Design Example #2—Software Serial I/O

An exercise often imposed on beginning microcomputer students is to write a program simulating a UART. Though doing this with the 8051 Family may appear to be a moot point (given that the hardware for a full UART is on-chip), it is still instructive to see how it would be done, and maintains a product line tradition.

As it turns out, the 8051 microcomputers can receive or transmit serial data via software very efficiently using the Boolean instruction set. Since any I/O pin may be a serial input or output, several serial links could be maintained at once.

Figures 12a and 12b show algorithms for receiving or transmitting a byte of data. (Another section of program would invoke this algorithm eight times, synchronizing it with a start bit, clock signal, software delay, or timer interrupt.) Data is received by testing an input pin, setting the carry to the same state, shifting the carry into a data buffer, and saving the partial frame in internal RAM. Data is transmitted by shifting an output buffer through the carry, and generating each bit on an output pin.

A side-by-side comparison of the software for this common “bit-banging” application with three different microprocessor architectures is shown in Table 5a and 5b. The 8051 solution is more efficient than the others on every count!
a.) Reception

b.) Transmission

Figure 12. Serial I/O Algorithms
Table 5. Serial I/O Programs for Various Microprocessors

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Input Routine</th>
<th>Output Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>8085</td>
<td>8048</td>
<td>8051</td>
</tr>
<tr>
<td>IN SERPORT</td>
<td>INI</td>
<td>MOV $SERPIN</td>
</tr>
<tr>
<td>ANI MASK</td>
<td>JNZ</td>
<td>MOV A SERBUF</td>
</tr>
<tr>
<td>JZ I O</td>
<td>JNZ110</td>
<td>RRC A</td>
</tr>
<tr>
<td>CMC C</td>
<td>CJP C</td>
<td>MOV SERBUF,A</td>
</tr>
<tr>
<td>LO CSI</td>
<td>MOV R0,$SERBUF</td>
<td></td>
</tr>
<tr>
<td>MOV A M</td>
<td>RRC A</td>
<td></td>
</tr>
<tr>
<td>MOV M A</td>
<td>MOV @R0,A</td>
<td></td>
</tr>
<tr>
<td>RESULTS:</td>
<td>RESULTS:</td>
<td>RESULTS:</td>
</tr>
<tr>
<td>10 INSTRUCTIONS</td>
<td>8 INSTRUCTIONS</td>
<td>4 INSTRUCTIONS</td>
</tr>
<tr>
<td>72 Bytes</td>
<td>9 BYTES</td>
<td>7 BYTES</td>
</tr>
<tr>
<td>19 uSEC.</td>
<td>22.5 uSEC.</td>
<td>4 uSEC.</td>
</tr>
<tr>
<td>b.) Output Routine.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8051</td>
<td>8048</td>
<td>8051</td>
</tr>
<tr>
<td>I XI HI $SERBUF</td>
<td>MOV R0,$SERBUF</td>
<td>MOV A SERBUF</td>
</tr>
<tr>
<td>MOV A M</td>
<td>MOV A @R0</td>
<td>RRC A</td>
</tr>
<tr>
<td>RR</td>
<td>RRC A</td>
<td>MOV SERBUF,A</td>
</tr>
<tr>
<td>MOV M A</td>
<td>MOV @R0,A</td>
<td></td>
</tr>
<tr>
<td>IN SERPORT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JC HI</td>
<td>JC HI</td>
<td></td>
</tr>
<tr>
<td>I O ANI NOT MASK</td>
<td>ANI SERPRTN NOT MASK</td>
<td>MOV SERPIN,C</td>
</tr>
<tr>
<td>JMP CNT</td>
<td>JMP CNT</td>
<td></td>
</tr>
<tr>
<td>HE ORI MASK</td>
<td>HE ORI SERPRTN</td>
<td>MASK</td>
</tr>
<tr>
<td>CN1 OUT SERPORT</td>
<td>CN1</td>
<td></td>
</tr>
<tr>
<td>RESULTS:</td>
<td>RESULTS:</td>
<td>RESULTS:</td>
</tr>
<tr>
<td>10 INSTRUCTIONS</td>
<td>8 INSTRUCTIONS</td>
<td>4 INSTRUCTIONS</td>
</tr>
<tr>
<td>72 Bytes</td>
<td>13 Bytes</td>
<td>7 BYTES</td>
</tr>
<tr>
<td>24 uSEC.</td>
<td>27.5 uSEC.</td>
<td>5 uSEC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design Example #3—Combinatorial Logic Equations

Next we'll look at some simple uses for bit-test instructions and logical operations. (This example is also presented in Application Note AP-69.)

Virtually all hardware designers have solved complex functions using combinatorial logic. While the hardware involved may vary from relay logic, vacuum tubes, or TTL to more esoteric technologies like fluidics, in each case the goal is the same: to solve a problem represented by a logical function of several Boolean variables.

Figure 13 shows TTL and relay logic diagrams for a function of the six variables U through Z. Each is a solution of the equation.

\[ Q = (U \cdot (V + W)) \cdot (X \cdot \overline{Y}) + Z \]

Equations of this sort might be reduced using Karnaugh Maps or algebraic techniques, but that is not the purpose of this example. As the logic complexity increases, so does the difficulty of the reduction process. Even a minor change to the function equations as the design evolves would require tedious re-reduction from scratch.
For the sake of comparison we will implement this function three ways, restricting the software to three proper subsets of the MCS-51 instruction set. We will also assume that U and V are input pins from different input ports, W and X are status bits for two peripheral controllers, and Y and Z are software flags set up earlier in the program. The end result must be written to an output pin on some third port. The first two implementations follow the flow-chart shown in Figure 14. Program flow would embark on a route down a test-and-branch tree and leaves either the “True” or “Not True” exit ASAP—as soon as the proper result has been determined. These exits then rewrite the output port with the result bit respectively one or zero.
Other digital computers must solve equations of this type with standard word-wide logical instructions and conditional jumps. So for the first implementation, we won't use any generalized bit-addressing instructions. As we shall soon see, being constrained to such an instruction subset produces somewhat sloppy software solutions. MCS-51 mnemonics are used in Example 2a: other machines might further cloud the situation by requiring operation-specific mnemonics like INPUT, OUTPUT, LOAD, STORE, etc., instead of the MOV mnemonic used for all variable transfers in the 8051 instruction set.

The code which results is cumbersome and error prone. It would be difficult to prove whether the software worked for all input combinations in programs of this sort. Furthermore, execution time will vary widely with input data.

Thanks to the direct bit-test operations, a single instruction can replace each move mask conditional jump sequence in Example 2a, but the algorithm would be equally convoluted (see Example 2b). To lessen the confusion “a bit” each input variable is assigned a symbolic name.

A more elegant and efficient implementation (Example 2c) strings together the Boolean ANL and ORL functions to generate the output function with straight-line code. When finished, the carry flag contains the result, which is simply copied out to the destination pin. No flow chart is needed—code can be written directly from the logic diagrams in Figure 14. The result is simplicity itself: fast, flexible, reliable, easy to design, and easy to debug.

An 8051 program can simulate an N-input AND or OR gate with at most N + 1 lines of source program—one for each input and one line to store the results. To simulate NAND and NOR gates, complement the carry after computing the function. When some inputs to the gate have “inversion bubbles”, perform the ANL or ORL operation on inverted operands. When the first input is inverted, either load the operand into the carry and then complement it, or use DeMorgan’s Theorem to convert the gate to a different form.

Example 2. Software Solutions to Logic Function of Figure 13.

a.) Using only byte-wide logical instructions

```
:BFUNCI SOLVE RANDOM LOGIC
   FUNCTION OF 6 VARIABLES
   BY LOADING AND MASKING
   THE APPROPRIATE BITS IN
   THE ACCUMULATOR. THEN
   EXECUTING CONDITIONAL
   JUMPS BASED ON ZERO
   CONDITION. (APPROACH USED
   BY BYTE-ORIENTED
   ARCHITECTURES.) BYTE AND
   MASK VALUES CORRESPOND TO
   RESPECTIVE BYTE ADDRESS
   AND BIT POSITIONS.

OUTBUF DATA 22H
:OUTPUT PIN STATE MAP
```
b.) Using only bit-test instructions

```assembly
:FUNCTION2 SOLVE A RANDOM LOGIC
: FUNCTION OF 6 VARIABLES
: BY DIRECTLY POLLING EACH BIT. (APPROACH USING
: MCS-51 UNIQUE BIT-TEST INSTRUCTION CAPABILITY.)
: SYMBOLS USED IN LOGIC DIAGRAM ASSIGNED TO
: CORRESPONDING 8x51 BIT ADDRESSES.
```

```
TESTV: MOV A,P2
ANL A,#00000100B
JNZ TESTU
MOV A,TCON
ANL A,#00100000B
JZ TESTX

TESTU: MOV A,P1
ANL A,#00000010B
JNZ SETQ

TESTX: MOV A,TCON
ANL A,#00010000B
JZ TESTZ
MOV A,20H
ANL A,#00000001B
JZ SETQ

TESTZ: MOV A,21H
ANL A,#00000010B
JZ SETQ
CLR Q: MOV A,OUTBUF
ANL A,#11110111B
JMP OUTQ
SETQ: MOV A,OUTBUF
ORL A,#00001000B
OUTQ: MOV OUTBUF,A
MOV P3,A
```

```
U  BIT  P1.1
V  BIT  P2.2
W  BIT  TFO
X  BIT  IE1
Y  BIT  20H.0
Z  BIT  21H.1
Q  BIT  P3.3

:FUNCTION2 SOLVE A RANDOM LOGIC
: FUNCTION OF 6 VARIABLES
: BY DIRECTLY POLLING EACH BIT. (APPROACH USING
: MCS-51 UNIQUE BIT-TEST INSTRUCTION CAPABILITY.)
: SYMBOLS USED IN LOGIC DIAGRAM ASSIGNED TO
: CORRESPONDING 8x51 BIT ADDRESSES.
```
An upper-limit can be placed on the complexity of software to simulate a large number of gates by summing the total number of inputs and outputs. The actual total should be somewhat shorter, since calculations can be “chained,” as shown. The output of one gate is often the first input to another, bypassing the intermediate variable to eliminate two lines of source.

Design Example #4—Automotive Dashboard Functions

Now let’s apply these techniques to designing the software for a complete controller system. This application is patterned after a familiar real-world application which isn’t nearly as trivial as it might first appear: automobile turn signals.

Imagine the three position turn lever on the steering column as a single-pole, triple-throw toggle switch. In its central position all contacts are open. In the up or down positions contacts close causing corresponding lights in the rear of the car to blink. So far very simple.

Two more turn signals blink in the front of the car, and two others in the dashboard. All six bulbs flash when an emergency switch is closed. A thermo-mechanical relay (accessible under the dashboard in case it wears out) causes the blinking.

Applying the brake pedal turns the tail light filaments on constantly ... unless a turn is in progress, in which case the blinking tail light is not affected. (Of course, the front turn signals and dashboard indicators are not affected by the brake pedal.) Table 6 summarizes these operating modes.

### Table 6. Truth Table for Turn-Signal Operation

<table>
<thead>
<tr>
<th>Brake Switch</th>
<th>Emerg. Switch</th>
<th>Left Turn Switch</th>
<th>Right Turn Switch</th>
<th>Left Front &amp; Dash</th>
<th>Right Front &amp; Dash</th>
<th>Left Rear</th>
<th>Right Rear</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Off</td>
<td>Blink</td>
<td>Off</td>
<td>Blink</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
<td>Blink</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
<td>On</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Off</td>
<td>Blink</td>
<td>On</td>
<td>Blink</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Blink</td>
<td>Off</td>
<td>On</td>
<td>On</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Blink</td>
<td>Blink</td>
<td>On</td>
<td>On</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Blink</td>
<td>Blink</td>
<td>On</td>
<td>Blink</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Blink</td>
<td>Blink</td>
<td>On</td>
<td>On</td>
</tr>
</tbody>
</table>
But we’re not done yet. Each of the exterior turn signal (but not the dashboard) bulbs has a second, somewhat dimmer filament for the parking lights. Figure 15 shows TTL circuitry which could control all six bulbs. The signals labeled “High Freq.” and “Low Freq.” represent two square-wave inputs. Basically, when one of the turn switches is closed or the emergency switch is activated the low frequency signal (about 1 Hz) is gated through to the appropriate dashboard indicator(s) and turn signal(s). The rear signals are also activated when the brake pedal is depressed provided a turn is not being made in the same direction. When the parking light switch is closed the higher frequency oscillator is gated to each front and rear turn signal, sustaining a low-intensity background level. (This is to eliminate the need for additional parking light filaments.)

In most cars, the switching logic to generate these functions requires a number of multiple-throw contacts. As many as 18 conductors thread the steering column of some automobiles solely for turn-signal and emergency blinker functions. (The author discovered this recently to his astonishment and dismay when replacing the whole assembly because of one burned contact.)

A multiple-conductor wiring harness runs to each corner of the car, behind the dash, up the steering column, and down to the blinker relay below. Connectors at each termination for each filament lead to extra cost and labor during construction, lower reliability and safety, and more costly repairs. And considering the system’s present complexity, increasing its reliability or detecting failures would be quite difficult.

There are two reasons for going into such painful detail describing this example. First, to show that the messiest part of many system designs is determining what the controller should do. Writing the software to solve these functions will be comparatively easy. Secondly, to show the many potential failure points in the system. Later we’ll see how the peripheral functions and intelligence built into a microcomputer (with a little creativity) can greatly reduce external interconnections and mechanical part count.

The Single-Chip Solution

The circuit shown in Figure 16 indicates five input pins to the five input variables—left-turn select, right-turn select, brake pedal down, emergency switch on, and parking lights on. Six output pins turn on the front, rear, and dashboard indicators for each side. The microcomputer implements all logical functions through software, which periodically updates the output signals as time elapses and input conditions change.

Figure 15. TTL Logic Implementation of Automotive Turn Signals

2-55
Design Example #3 demonstrated that symbolic addressing with user-defined bit names makes code and documentation easier to write and maintain. Accordingly, we'll assign these I/O pins names for use throughout the program. (The format of this example will differ somewhat from the others. Segments of the overall program will be presented in sequence as each is described.)

```
INPUT PIN DECLARATIONS:
:(ALL INPUTS ARE POSITIVE-TRUE LOGIC)
:  BRAKE BIT P1.0 ;BRAKE PEDAL ;DEPRESSED
  EMERG BIT P1.1 ;EMERGENCY BLINKER ;ACTIVATED
  PARK BIT P1.2 ;PARKING LIGHTS ON
  I_TURN BIT P1.3 ;TURN LEVER DOWN
  R_TURN BIT P1.4 ;TURN LEVER UP

OUTPUT PIN DECLARATIONS:
:  L.FRNT BIT P1.5 ;FRONT LEFT-TURN ;INDICATOR
    R.FRNT BIT P1.6 ;FRONT RIGHT-TURN ;INDICATOR
    L.DASH BIT P1.7 ;DASHBOARD LEFT-TURN ;INDICATOR
    R_DASH BIT P2.0 ;DASHBOARD RIGHT-TURN ;INDICATOR
    L_REAR BIT P2.1 ;REAR LEFT-TURN ;INDICATOR
    R_REAR BIT P2.2 ;REAR RIGHT-TURN ;INDICATOR
```

Another key advantage of symbolic addressing will appear further on in the design cycle. The locations of cable connectors, signal conditioning circuitry, voltage regulators, heat sinks, and the like all affect P.C. board layout. It's quite likely that the somewhat arbitrary pin assignment defined early in the software design cycle will prove to be less than optimum; rearranging the I/O pin assignment could well allow a more compact module, or eliminate costly jumpers on a single-sided board. (These considerations apply especially to automotive and other cost-sensitive applications needing single-chip controllers.) Since other architectures mask bytes or use "clever" algorithms to isolate bits by rotating them into the carry, re-routing an input signal (from bit 1 of port 1, for example, to bit 4 of port 3) could require extensive modifications throughout the software.

The Boolean Processor's direct bit addressing makes such changes absolutely trivial. The number of the port containing the pin is irrelevant, and masks and complex
program structures are not needed. Only the initial Boolean variable declarations need to be changed; ASM51 automatically adjusts all addresses and symbolic references to the reassigned variables. The user is assured that no additional debugging or software verification will be required.

```
:  ; INTERRUPT RATE SUBDIVIDER
SUB_DIV DATA 20H
; HIGH-FREQUENCY OSCILLATOR BIT
HI_FREQ BIT SUB_DIV,0
; LOW-FREQUENCY OSCILLATOR BIT
LO_FREQ BIT SUB_DIV,7
:  ; INTERRUPT RATE SUBDIVIDER
ORG 0000H
JMP INIT
:  ; INTERRUPT RATE SUBDIVIDER
ORG 100H
; PUT TIMER 0 IN MODE 1
INIT:  MOV TMOD,#0000001B
; INITIALIZE TIMER REGISTERS
MOV TL0,#0
MOV TH0,#-16
; SUBDIVIDE INTERRUPT RATE BY 244
MOV SUB_DIV,#244
; ENABLE TIMER INTERRUPTS
SETB ET0
; GLOBALLY ENABLE ALL INTERRUPTS
SETB EA
; START TIMER
SETB TRO
;
; (CONTINUE WITH BACKGROUND PROGRAM)
;
; PUT TIMER 0 IN MODE 1
; INITIALIZE TIMER REGISTERS
; SUBDIVIDE INTERRUPT RATE BY 244
; ENABLE TIMER INTERRUPTS
; GLOBALLY ENABLE ALL INTERRUPTS
; START TIMER
```

Timer 0 (one of the two on-chip timer counters) replaces the thermo-mechanical blinker relay in the dash-board controller. During system initialization it is configured as a timer in mode 1 by setting the least significant bit of the timer mode register (TMOD). In this configuration the low-order byte (TL0) is incremented every machine cycle, overflowing and incrementing the high-order byte (TH0) every 256 μs. Timer interrupt 0 is enabled so that a hardware interrupt will occur each time TH0 overflows.

An eight-bit variable in the bit-addressable RAM array will be needed to further subdivide the interrupts via software. The lowest-order bit of this counter toggles very fast to modulate the parking lights: bit 7 will be “tuned” to approximately 1 Hz for the turn- and emergency-indicator blinking rate.

Loading TH0 with -16 will cause an interrupt after 4.096 ms. The interrupt service routine reloads the high-order byte of timer 0 for the next interval, saves the CPU registers likely to be affected on the stack, and then decrements SUB_DIV. Loading SUB_DIV with 244 initially and each time it decrements to zero will produce a 0.999 second period for the highest-order bit.

```
ORG 000BH ; TIMER 0 SERVICE VECTOR
MOV TH0,#-16
PUSH PSW
PUSH ACC
PUSH B
DJNZ SUB_DIV,TOSERV
MOV SUB_DIV,#244
```

The code to sample inputs, perform calculations, and update outputs—the real “meat” of the signal controller algorithm—may be performed either as part of the interrupt service routine or as part of a background program loop. The only concern is that it must be executed at least several dozen times per second to prevent parking light flickering. We will assume the former case, and insert the code into the timer 0 service routine.

First, notice from the logic diagram (Figure 15) that the subterm (PARK • H_FREQ), asserted when the parking lights are to be on dimly, figures into four of the six output functions. Accordingly, we will first compute that term and save it in a temporary location named “DIM”. The PSW contains two general purpose flags: FO, which corresponds to the 8048 flag of the same name, and PSW.1. Since the PSW has been saved and will be restored to its previous state after servicing the interrupt, we can use either bit for temporary storage.

```
DIM BIT PSW.1 ; DECLARE TEMP ; STORAGE FLAG
; ...
MOV C,PARK ; GATE PARKING
ANL HI_FREQ ; WITH HIGH ; FREQUENCY ; SIGNAL
MOV DIM,C ; AND SAVE IN ; TEMP. VARIABLE
```

This simple three-line section of code illustrates a remarkable point. The software indicates in very abstract terms exactly what function is being performed, inde-
dependent of the hardware configuration. The fact that these three bits include an input pin, a bit within a program variable, and a software flag in the PSW is totally invisible to the programmer.

Now generate and output the dashboard left turn signal.

```
; MOV C,L_TURN ;SET CARRY IF ;TURN
ORL C,EMERG ;OR EMERGENCY ;SELECTED
ANL C,LO_FREQ ;GATE IN 1 Hz ;SIGNAL
MOV L_DASH,C ;AND OUTPUT TO ;DASHBOARD
```

To generate the left front turn signal we only need to add the parking light function in FO. But notice that the function in the carry will also be needed for the rear signal. We can save effort later by saving its current state in FO.

```
; MOV FO,C ;SAVE FUNCTION ;SO FAR
ORL C,DIM ;ADD IN PARKING ;LIGHT FUNCTION
MOV L_FRNT,C ;AND OUTPUT TO ;TURN SIGNAL
```

Finally, the rear left turn signal should also be on when the brake pedal is depressed, provided a left turn is not in progress.

```
MOV C,BRAKE ;GATE BRAKE
ANL C,L_TURN ;WITH TURN ;LEVER
ORL C,FO ;INCLUDE TEMP. ;VARIABLE FROM DASH
```

Now we have to go through a similar sequence for the right-hand equivalents to all the left-turn lights. This also gives us a chance to see how the code segments above look when combined.

```
; MOV C,R_TURN ;SET CARRY H- ;TURN
ORL C,EMERG ;OR EMERGENCY ;SELECTED
ANL C,LO_FREQ ;IF SO, GATE IN 1 Hz SIGNAL
MOV R_DASH,C ;AND OUTPUT TO ;DASHBOARD
MOV FO,C ;SAVE FUNCTION ;SO FAR
ORL C,DIM ;ADD IN PARKING, ;LIGHT FUNCTION
MOV R_FRNT,C ;AND OUTPUT TO ;TURN SIGNAL
MOV C.BRAKE ;GATE BRAKE ;PEDAL SWITCH
ANL C.R_TURN ;WITH TURN ;LEVER
ORL C.FO ;INCLUDE TEMP. ;VARIABLE FROM ;DASH
ORL C,DIM ;AND PARKING ;LIGHT FUNCTION
MOV R_REAR,C ;AND OUTPUT TO ;TURN SIGNAL
```

(The perceptive reader may notice that simply rearranging the steps could eliminate one instruction from each sequence.)

Now that all six bulbs are in the proper states, we can return from the interrupt routine, and the program is finished. This code essentially needs to reverse the status saving steps at the beginning of the interrupt.

<table>
<thead>
<tr>
<th>Sub_Div Bits</th>
<th>Duty Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>12.5% 25.0% 37.5% 50.0% 62.5% 75.0% 87.5%</td>
</tr>
<tr>
<td>X X X X X X 0 0</td>
<td>Off Off Off Off Off Off Off</td>
</tr>
<tr>
<td>X X X X X X 0 0</td>
<td>Off Off Off Off Off Off On</td>
</tr>
<tr>
<td>X X X X X X 0 1</td>
<td>Off Off Off Off Off Off On</td>
</tr>
<tr>
<td>X X X X X X 0 1</td>
<td>Off Off Off Off Off On On</td>
</tr>
<tr>
<td>X X X X X X 1 0</td>
<td>Off Off Off Off Off On On</td>
</tr>
<tr>
<td>X X X X X X 1 0</td>
<td>Off Off Off Off On On On</td>
</tr>
<tr>
<td>X X X X X X 1 1</td>
<td>Off Off Off On On On On</td>
</tr>
<tr>
<td>X X X X X X 1 1</td>
<td>Off Off On On On On On</td>
</tr>
<tr>
<td>X X X X X X 1 1</td>
<td>Off On On On On On On</td>
</tr>
<tr>
<td>X X X X X X 1 1</td>
<td>Off On On On On On On</td>
</tr>
</tbody>
</table>
Program Refinements. The luminescence of an incandescent light bulb filament is generally non-linear: the 50% duty cycle of HI_FREQ may not produce the desired intensity. If the application requires, duty cycles of 25%, 75%, etc. are easily achieved by ANDing and OR'ing in additional low-order bits of SUB_DIV. For example, 30 H/ signals of seven different duty cycles could be produced by considering bits 2-0 as shown in Table 7. The only software change required would be to the code which sets-up variable DIM:

\[
\begin{align*}
\text{MOV } C, \text{SUBDIV.1} &; \text{START WITH 50 PERCENT} \\
\text{ANL } C, \text{SUBDIV.0} &; \text{MASK DOWN TO 25 PERCENT} \\
\text{ORL } C, \text{SUBDIV.2} &; \text{AND BUILD BACK TO 62 PERCENT} \\
\text{MOV } \text{DIM}, C &; \text{DUTY CYCLE FOR PARKING LIGHTS.}
\end{align*}
\]

Interconnections increase cost and decrease reliability. The simple buffered pin-per-function circuit in Figure 16 is insufficient when many outputs require higher-than-TTL drive levels. A lower-cost solution uses the 8051 serial port in the shift-register mode to augment I/O. In mode 0, writing a byte to the serial port data buffer (SBUF) causes the data to be output sequentially through the "RXD" pin while a burst of eight clock pulses is generated on the "TXD" pin. A shift register connected to these pins (Figure 17) will load the data byte as it is shifted out. A number of special peripheral driver circuits combining shift-register inputs with high drive level outputs have been introduced recently.

Cascading multiple shift registers end-to-end will expand the number of outputs even further. The data rate in the I/O expansion mode is one megabaud, or 8 μs per byte. This is the mode which the serial port defaults to following a reset, so no initialization is required.

The software for this technique uses the B register as a "map" corresponding to the different output functions. The program manipulates these bits instead of the output pins. After all functions have been calculated the B register is shifted by the serial port to the shift-register driver. (While some outputs may glitch as data is shifted through them, at 1 Megabaud most people wouldn't notice. Some shift registers provide an "enable" bit to hold the output states while new data is being shifted in.)

This is where the earlier decision to address bits symbolically throughout the program is going to pay off. This major I/O restructuring is nearly as simple to implement as rearranging the input pins. Again, only the bit declarations need to be changed.

\[
\begin{align*}
\text{I_FRTN BIT } B.0 &; \text{FRONT LEFT-TURN INDICATOR} \\
\text{R_FRTN BIT } B.1 &; \text{FRONT RIGHT-TURN INDICATOR} \\
\text{I_DASH BIT } B.2 &; \text{DASHBOARD LEFT-TURN INDICATOR} \\
\text{R_DASH BIT } B.3 &; \text{DASHBOARD RIGHT-TURN INDICATOR} \\
\text{I_REAR BIT } B.4 &; \text{REAR LEFT-TURN INDICATOR} \\
\text{R_REAR BIT } B.5 &; \text{REAR RIGHT-TURN INDICATOR}
\end{align*}
\]

---

**Figure 17. Output Expansion Using Serial Port**

2-59
The original program to compute the functions need not change. After computing the output variables, the control map is transmitted to the buffered shift register through the serial port.

```
MOV SBUF,B ;LOAD BUFFER AND TRANSMIT
```

The Boolean Processor solution holds a number of advantages over older methods. Fewer switches are required. Each is simpler, requiring fewer poles and lower current contacts. The flasher relay is eliminated entirely. Only six filaments are driven, rather than 10. The wiring harness is therefore simpler and less expensive—one conductor for each of the six lamps and each of the five sensor switches. The fewer conductors use far fewer connectors. The whole system is more reliable.

And since the system is much simpler it would be feasible to implement redundancy and or fault detection on the four main turn indicators. Each could still be a standard double filament bulb, but with the filaments driven in parallel to tolerate single-element failures.

Even with redundancy, the lights will eventually fail. To handle this inescapable fact current or voltage sensing circuits on each main drive wire can verify that each bulb and its high-current driver is functioning properly. Figure 18 shows one such circuit.

Assume all of the lights are turned on except one: i.e., all but one of the collectors are grounded. For the bulb which is turned off, if there is continuity from +12V through the bulb base and filament, the control wire, all connectors, and the P.C. board traces, and if the transistor is indeed not shorted to ground, then the collector will be pulled to +12V. This turns on the base of Q8 through the corresponding resistor, and grounds the input pin, verifying that the bulb circuit is operational. The continuity of each circuit can be checked by software in this way.

![Figure 18 Wiring Harness Diagram](image-url)
Now turn all the bulbs on, grounding all the collectors. Q7 should be turned off, and the Test pin should be high. However, a control wire shorted to +12V or an open-circuited drive transistor would leave one of the collectors at the higher voltage even now. This too would turn on Q7, indicating a different type of failure. Software could perform these checks once per second by executing the routine every time the software counter SUB_DIV is reloaded by the interrupt routine.

```
DJNZ SUB_DIV, TOSERV
MOV SUB_DIV, #244 ; RELOAD COUNTER
ORL P1, #1110000B ; SET CONTROL
ORL P2, #0000111B ; OUTPUTS HIGH
CLR L_FRNT ; FLOAT DRIVE
CLR L_DASH ; COLLECTOR
JB TO, FAULT ; TO SHOULD BE FULLY LOW
SETB L_FRNT ; FULL COLLECTOR
SETB L_DASH ; BACK DOWN
CLR L_REAR
JB TO, FAULT
SETB L_REAR
CLR R_FRNT
JB TO, FAULT
SETB R_FRNT
CLR R_DASH
JB TO, FAULT
SETB R_DASH
CLR R_REAR
JB TO, FAULT
SETB R_REAR

; WITH ALL COLLECTORS GROUNDED, TO SHOULD BE HIGH
; IF SO, CONTINUE WITH INTERRUPT ROUTINE.
JB TO, TOSERV
FAULT: ; ELECTRICAL
; FAILURE
; PROCESSING
; ROUTINE
; (LEFT TO
; READER'S
; IMAGINATION)
TOSERV: ; CONTINUE WITH
; INTERRUPT
; PROCESSING

```

The complete assembled program listing is printed in Appendix A. The resulting code consists of 67 program statements, not counting declarations and comments, which assemble into 150 bytes of object code. Each pass through the service routine requires (coincidentally) 67 µs plus 32 µs once per second for the electrical test. If executed every 4 ms as suggested this software would typically reduce the throughput of the background program by less than 2%.

Once a microcomputer has been designed into a system, new features suddenly become virtually free. Software could make the emergency blinkers flash alternately or at a rate faster than the turn signals. Turn signals could override the emergency blinkers. Adding more bulbs would allow multiple tail light sequencing and syncopation—true flash factor, so to speak.

Design Example #5—Complex Control Functions

Finally, we'll mix byte and bit operations to extend the use of 8051 into extremely complex applications.

Programmers can arbitrarily assign I/O pins to input and output functions only if the total does not exceed 32, which is insufficient for applications with a very large number of input variables. One way to expand the number of inputs is with a technique similar to multiplexed-keyboard scanning.

Figure 19 shows a block diagram for a moderately complex programmable industrial controller with the following characteristics:

- 64 input variable sensors:
- 12 output signals:
- Combinational and sequential logic computations:
- Remote operation with communications to a host processor via a high-speed full-duplex serial link:
- Two prioritized external interrupts:
- Internal real-time and time-of-day clocks.

While many microprocessors could be programmed to provide these capabilities with assorted peripheral support chips, an 8051 microcomputer needs no other integrated circuits!

The 64 input sensors are logically arranged as an 8x8 matrix. The pins of Port 1 sequentially enable each column of the sensor matrix: as each is enabled Port 0 reads in the state of each sensor in that column. An eight-byte block in bit-addressable RAM remembers the data as it is read in so that after each complete scan cycle there is an internal map of the current state of all sensors. Logic functions can then directly address the elements of the bit map.
The computer's serial port is configured as a nine-bit UART, transferring data at 17,000 bytes-per-second. The ninth bit may distinguish between address and data bytes.

The 8051 serial port can be configured to detect bytes with the address bit set, automatically ignoring all others. Pins INT0 and INT1 are interrupts configured respectively as high-priority, falling-edge triggered and low-priority, low-level triggered. The remaining 12 I/O pins output TTL-level control signals to 12 actuators.

There are several ways to implement the sensor matrix circuitry, all logically similar. Figure 20a shows one possibility. Each of the 64 sensors consists of a pair of simple switch contacts in series with a diode to permit multiple contact closures throughout the matrix.

The scan lines from Port 1 provide eight un-encoded active-high scan signals for enabling columns of the matrix. The return lines on rows where a contact is closed are pulled high and read as logic ones. Open return lines are pulled to ground by one of the 40 kΩ resistors and are read as zeroes. (The resistor values must be chosen to ensure all return lines are pulled above the 2.0V logic threshold, even in the worst-case,
where all contacts in an enabled column are closed.) Since P0 is provided open-collector outputs and high-impedance MOS inputs its input loading may be considered negligible.

The circuits in Figures 20b–20d are variations on this theme. When input signals must be electrically isolated from the computer circuitry as in noisy industrial environments, phototransistors can replace the switch diode pairs and provide optical isolation as in Figure 20b. Additional opto-isolators could also be used on the control output and special signal lines.

The other circuits assume that input signals are already at TTL levels. Figure 20c uses octal three-state buffers enabled by active-low scan signals to gate eight signals onto Port 0. Port 0 is available for memory expansion or peripheral chip interfacing between sensor matrix scans. Eight-to-one multiplexers in Figure 20d select one of eight inputs for each return line as determined by encoded address bits output on three pins of Port 1. (Five more output pins are thus freed for more control functions.) Each output can drive at least one standard TTL or up to 10 low-power TTL loads without additional buffering.

Going back to the original matrix circuit, Figure 21 shows the method used to scan the sensor matrix. Two complete bit maps are maintained in the bit-addressable region of the RAM: one for the current state and one for the previous state read for each sensor. If the need arises, the program could then sense input transitions and or debounce contact closures by comparing each bit with its earlier value.

The code in Example 3 implements the scanning algorithm for the circuits in Figure 20a. Each column is enabled by setting a single bit in a field of zeroes. The bit maps are positive logic: ones represent contacts that are closed or isolators turned on.

Example 3.

```
Example 3.
INPUT_SCAN: ;SUBROUTINE TO READ
;CURRENT STATE
;OF 64 SENSORS AND
;SAVE IN RAM 20H-27H
  MOV R0,#20H ;INITIALIZE
  MOV R1,#28H ;FOR BIT MAP
  MOV A,#80H
  SCAN: MOV PI,A
  RR A
  MOV R2,A
  MOV A,PO
  XCH A.@RO
  MOV @RI,A
  INC RO
  INC RI
  MOV A,R2
  BASES
  OUTPUT TO SCAN
  RR A
  SHIFT TO ENABLE
  NEXT COLUMN
  NEXT
  MOV R2,A
  REMEMBER CURRENT
  SCAN
  POSITION
  READ RETURN
  LINES
  SWITCH WITH
  PREVIOUS MAP
  BITS
  SAVE PREVIOUS
  STATE AS WELL
  BUMP POINTERS
  RELOAD SCAN
  LINE MASK
  LOOP UNTIL ALL
  EIGHT COLUMNS
  READ
```

2-63
Figure 20. Sensor Matrix Implementation Methods
b.) Using Optically-Coupled Isolators

Figure 20. Sensor Matrix Implementation Methods (Continued)
c.) Using TTL Three-State Buffers

Figure 20. Sensor Matrix Implementation Methods (Continued)
d. Using TTL Data Selectors

Figure 20. Sensor Matrix Implementation Methods (Continued)
What happens after the sensors have been scanned depends on the individual application. Rather than inventing some artificial design problem, software corresponding to commonplace logic elements will be discussed.

**Combinatorial Output Variables.** An output variable which is a simple (or not so simple) combinational function of several input variables is computed in the spirit of Design Example 3. All 64 inputs are represented in the bit maps: in fact, the sensor numbers in Figure 20 correspond to the absolute bit addresses in RAM! The code in Example 4 activates an actuator connected to P2.2 when sensors 12, 23, and 34 are closed and sensors 45 and 56 are open.

**Example 4.**
Simple Combinatorial Output Variables.

\[
\text{SET } P2.2 = (12)(23)(34)(45)(56)
\]
\[
\text{MOV } C, 12 \\
\text{ANL } C, 23 \\
\text{ANL } C, 34 \\
\text{ANL } C, 45 \\
\text{ANL } C, 56 \\
\text{MOV } P2.2, C
\]

**Intermediate Variables.** The examination of a typical relay-logic ladder diagram will show that many of the rungs control not outputs but rather relays whose contacts figure into the computation of other functions. In effect, these relays indicate the state of intermediate variables of a computation.

The MCS-51 solution can use any directly addressable bit for the storage of such intermediate variables. Even when all 128 bits of the RAM array are dedicated (to input bit maps in this example), the accumulator, PSW, and B register provide 18 additional flags for intermediate variables.

For example, suppose switches 0 through 3 control a safety interlock system. Closing any of them should deactivate certain outputs. Figure 22 is a ladder diagram for this situation. The interlock function could be recomputed for every output affected, or it may be computed once and save (as implied by the diagram). As the program proceeds this bit can qualify each output.
Example 5. Incorporating Override signal into actuator outputs.

; CALL INPUT_SCAN
MOV C,0
ORL C,1
ORL C,2
ORL C,3
MOV FO,C
; .... ....
; COMPUTE FUNCTION 0
; ANL C, FO
MOV PLO,C
; .... ....
; COMPUTE FUNCTION 1
; ANL C, FO
MOV Pl,1,C
; .... ....
; COMPUTE FUNCTION 2
; ANL C, FO
MOV P1,2,C
; .... ....

_Latching Relays._ A latching relay can be forced into either the ON or OFF state by two corresponding input signals, where it will remain until forced onto the opposite state—analogous to a TTL Set/Reset flip-flop. The relay is used as an intermediate variable for other calculations. In the previous example, the emergency condition could be remembered and remain active until an "emergency cleared" button is pressed.

Any flag or addressable bit may represent a latching relay with a few lines of code (see Example 6).

Example 6. Simulating a latching relay.

;I_SET SET FLAG 0 IF C=1
I_SET: ORL C,FO
MOV FO,C
;
;I_RSET RESET FLAG 0 IF C=1
I_RSET: CPS C
ANL C,FO
MOV FO,C
;

_Time Delay Relays._ A time delay relay does not respond to an input signal until it has been present (or absent) for some predefined time. For example, a ballast or load resistor may be switched in series with a D.C. motor when it is first turned on, and shunted from the circuit after one second. This sort of time delay may be simulated by an interrupt routine driven by one of the two 8051 timer counters. The procedure followed by the routine depends heavily on the details of the exact function needed: time-outs or time delays with resettable or non-resettable inputs are possible. If the interrupt routine is executed every 10 milliseconds the code in Example 7 will clear an intermediate variable set by the background program after it has been active for two seconds.

Example 7. Code to clear USRFLG after a fixed time delay.

JNB USR_FLG,NXTTST
DJNZ DLAY_COUNT,NXTTST
CLR USR_FLG
MOV DLAY_COUNT,#200
NXTTST: ;.; ....

Figure 22. Ladder Diagram for Output Override Circuitry
Serial Interface to Remote Processor. When it detects emergency conditions represented by certain input combinations (such as the earlier Emergency Override), the controller could shut down the machine immediately and/or alert the host processor via the serial port. Code bytes indicating the nature of the problem could be transmitted to a central computer. In fact, at 17,000 bytes-per-second, the entire contents of both bit maps could be sent to the host processor for further analysis in less than a millisecond! If the host decides that conditions warrant, it could alert other remote processors in the system that a problem exists and specify which shut-down sequence each should initiate. For more information on using the serial port, consult the MCS-51 User’s Manual.

Response Timing

One difference between relay and programmed industrial controllers (when each is considered as a “black box”) is their respective reaction times to input changes. As reflected by a ladder diagram, relay systems contain a large number of “rungs” operating in parallel. A change in input conditions will begin propagating through the system immediately, possibly affecting the output state within milliseconds.

Software, on the other hand, operates sequentially. A change in input states will not be detected until the next time an input scan is performed, and will not affect the outputs until that section of the program is reached. For that reason the raw speed of computing the logical functions is of extreme importance.

Here the Boolean processor pays off. Every instruction mentioned in this Note completes in one or two microseconds—the minimum instruction execution time for many other microcontrollers! A ladder diagram containing a hundred rungs, with an average of four contacts per rung can be replaced by approximately five hundred lines of software. A complete pass through the entire matrix scanning routine and all computations would require about a millisecond: less than the time it takes for most relays to change state.

A programmed controller which simulates each Boolean function with a subroutine would be less efficient by at least an order of magnitude. Extra software is needed for the simulation routines, and each step takes longer to execute for three reasons: several byte-wide logical instructions are executed per user program step (rather than one Boolean operation): most of those instructions take longer to execute with microprocessors performing multiple off-chip accesses: and calling and returning from the various subroutines requires overhead for stack operations.

In fact, the speed of the Boolean Processor solution is likely to be much faster than the system requires. The CPU might use the time left over to compute feedback parameters, collect and analyze execution statistics, perform system diagnostics, and so forth.

Additional Functions and Uses

With the building-block basics mentioned above many more operations may be synthesized by short instruction sequences.

Exclusive-OR. There are no common mechanical devices or relays analogous to the Exclusive-OR operation, so this instruction was omitted from the Boolean Processor. However, the Exclusive-OR or Exclusive-NOR operation may be performed in two instructions by conditionally complementing the carry or a Boolean variable based on the state of any other testable bit.

:XCH. The contents of the carry and some other bit may be exchanged (switched) by using the accumulator as temporary storage. Bits can be moved into and out of the accumulator simultaneously using the Rotate-
through-carry instructions, though this would alter the accumulator data.

```plaintext
;EXCHANGE CARRY WITH USRFLG
XCHBIT: RLC A
MOV C,USR_FLG
RRC A
MOV USR_FLG,C
RLC A
```

*Extended Bit Addressing.* The 8051 can directly address 144 general-purpose bits for all instructions in Figure 3b. Similar operations may be extended to any bit anywhere on the chip with some loss of efficiency.

The logical operations AND, OR, and Exclusive-OR are performed on byte variables using six different addressing modes, one of which lets the source be an immediate mask, and the destination any directly addressable byte. Any bit may thus be set, cleared, or complemented with a three-byte, two-cycle instruction if the mask has all bits but one set or cleared.

Byte variables, registers, and indirectly addressed RAM may be moved to a bit addressable register (usually the accumulator) in one instruction. Once transferred, the bits may be tested with a conditional jump, allowing any bit to be polled in 3 microseconds—still much faster than most architectures—or used for logical calculations. (This technique can also simulate additional bit addressing modes with byte operations.)

*Parity of bytes or bits.* The parity of the current accumulator contents is always available in the PSW, from whence it may be moved to the carry and further processed. Error-correcting Hamming codes and similar applications require computing parity on groups of isolated bits. This can be done by conditionally complementing the carry flag based on those bits or by gathering the bits into the accumulator (as shown in the DES example) and then testing the parallel parity flag.

*Multiple byte shift and CRC codes*

Though the 8051 serial port can accommodate eight- or nine-bit data transmissions, some protocols involve much longer bit streams. The algorithms presented in Design Example 2 can be extended quite readily to 16 or more bits by using multi-byte input and output buffers.

Many mass data storage peripherals and serial communications protocols include Cyclic Redundancy (CRC) codes to verify data integrity. The function is generally computed serially by hardware using shift registers and Exclusive-OR gates, but it can be done with software. As each bit is received into the carry, appropriate bits in the multi-byte data buffer are conditionally complemented based on the incoming data bit. When finished, the CRC register contents may be checked for zero by ORing the two bytes in the accumulator.

### 4.0 SUMMARY

A truly unique facet of the Intel MCS-51 microcomputer family design is the collection of features optimized for the one-bit operations so often desired in real-world, real-time control applications. Included are 17 special instructions, a Boolean accumulator, implicit and direct addressing modes, program and mass data storage, and many I/O options. These are the world's first single-chip microcomputers able to efficiently manipulate, operate on, and transfer either bytes or individual bits as data.

This Application Note has detailed the information needed by a microcomputer system designer to make full use of these capabilities. Five design examples were used to contrast the solutions allowed by the 8051 and those required by previous architectures. Depending on the individual application, the 8051 solution will be easier to design, more reliable to implement, debug, and verify, use less program memory, and run up to an order of magnitude faster than the same function implemented on previous digital computer architectures.

Combining byte- and bit-handling capabilities in a single microcomputer has a strong synergistic effect: the power of the result exceeds the power of byte- and bit-processors laboring individually. Virtually all user applications will benefit in some way from this duality. Data intensive applications will use bit addressing for test pin monitoring or program control flags: control applications will use byte manipulation for parallel I/O expansion or arithmetic calculations.

It is hoped that these design examples give the reader an appreciation of these unique features and suggest ways to exploit them in his or her own application.
THE FOLLOWING PROGRAM USES THE BOOLEAN INSTRUCTION SET
OF THE INTEL 8051 MICROCOMPUTER TO PERFORM A NUMBER OF
AUTOMOTIVE DASHBOARD CONTROL FUNCTIONS RELATING TO
TURN SIGNAL CONTROL, EMERGENCY BLINKERS, BRAKE LIGHT
CONTROL, AND PARKING LIGHT OPERATION.
THE ALGORITHMS AND HARDWARE ARE DESCRIBED IN DESIGN
EXAMPLE #4 OF INTEL APPLICATION NOTE AP-70,
"USING THE INTEL MCS-51(TM)
BOOLEAN PROCESSING CAPABILITIES"

INPUT PIN DECLARATIONS
(ALL INPUTS ARE POSITIVE-TRUE LOGIC.
INPUTS ARE HIGH WHEN RESPECTIVE SWITCH CONTACT IS CLOSED)

0090 BRAKE BIT P.0 BRAKE PEDAL DEPRESSED
0091 EMERG BIT P.1 EMERGENCY BLINKER ACTIVATED
0092 PARK BIT P.2 PARKING LIGHTS ON
0093 L_TURN BIT P.3 TURN LEVER DOWN
0094 R_TURN BIT P.4 TURN LEVER UP

OUTPUT PIN DECLARATIONS
(ALL OUTPUTS ARE POSITIVE TRUE LOGIC
BULB IS TURNED ON WHEN OUTPUT PIN IS HIGH)

0095 L_FRNT BIT P.5 FRONT LEFT-TURN INDICATOR
0096 R_FRNT BIT P.6 FRONT RIGHT-TURN INDICATOR
0097 L_DASH BIT P.7 DASHBOARD LEFT-TURN INDICATOR
0098 R_DASH BIT P.8 DASHBOARD RIGHT-TURN INDICATOR
0099 L_REAR BIT P.9 REAR LEFT-TURN INDICATOR
0100 R_REAR BIT P.10 REAR RIGHT-TURN INDICATOR
0101 S_FAIL BIT P.11 ELECTRICAL SYSTEM FAULT INDICATOR

INTERNAL VARIABLE DEFINITIONS

0020 SUB_DIV DATA 20H INTERRUPT-RATE SUBDIVIDER
0000 HI_FREQ BIT SUB_DIV 0 HIGH-FREQUENCY OSCILLATOR BIT
0007 LO_FREQ BIT SUB_DIV 7 LOW-FREQUENCY OSCILLATOR BIT
0011 DIM BIT PSW 1 PARKING LIGHTS ON FLAG

*EJECT
<table>
<thead>
<tr>
<th>LOC OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 020040</td>
<td>49</td>
<td>ORG O000H, RESET VECTOR</td>
</tr>
<tr>
<td>0000 020040</td>
<td>50</td>
<td>LJMP INIT</td>
</tr>
<tr>
<td>0008 75BCFO</td>
<td>52</td>
<td>ORG O000B, TIMER O SERVICE VECTOR</td>
</tr>
<tr>
<td>0008 75BCFO</td>
<td>53</td>
<td>MOV TH0. #16, HIGH TIMER BYTE ADJUSTED TO CONTROL INT RATE</td>
</tr>
<tr>
<td>000E C00D</td>
<td>54</td>
<td>PUSH PSW, EXECUTE CODE TO SAVE ANY REGISTERS USED BELOW</td>
</tr>
<tr>
<td>0010 0154</td>
<td>55</td>
<td>LJMP UPDATE, (CONTINUE WITH REST OF ROUTINE)</td>
</tr>
<tr>
<td>0040 75BA00</td>
<td>57</td>
<td>ORG O040H, ZERO LOADED INTO LOW-ORDER BYTE AND</td>
</tr>
<tr>
<td>0040 75BA00</td>
<td>58</td>
<td>INIT MOV TL0. #0, -16 IN HIGH-ORDER BYTE GIVES 4 MSEC PERIOD</td>
</tr>
<tr>
<td>0046 75B961</td>
<td>60</td>
<td>MOV TMOD. #01100001B, 8-BIT AUTO RELLOAD COUNTER MODE FORTIMER 1,</td>
</tr>
<tr>
<td>0046 75B961</td>
<td>61</td>
<td>16-BIT TIMER MODE FOR TIMER 0 SELECTED</td>
</tr>
<tr>
<td>0049 7520F4</td>
<td>62</td>
<td>MOV SUB_DIV. #244, SUBDIVIDE INTERRUPT RATE BY 244 FOR 1 HZ</td>
</tr>
<tr>
<td>004C D2AF</td>
<td>63</td>
<td>SETB ETO, USE TIMER 0 OVERFLOWS TO INTERRUPT PROGRAM</td>
</tr>
<tr>
<td>004F D2AF</td>
<td>64</td>
<td>SETB EA, CONFIGURE IE TO GLOBALLY ENABLE INTERRUPTS</td>
</tr>
<tr>
<td>0050 D28C</td>
<td>65</td>
<td>SETB TRO, KEEP INSTRUCTION CYCLE COUNT UNTIL OVERFLOW</td>
</tr>
<tr>
<td>0052 80FE</td>
<td>66</td>
<td>SJMP $, START BACKGROUND PROGRAM EXECUTION</td>
</tr>
<tr>
<td>0054 D5203B</td>
<td>69</td>
<td>UPDATE DJNZ SUB_DIV_TOOSERV, EXECUTE SYSTEM TEST ONLY ONCE PER SECOND</td>
</tr>
<tr>
<td>0057 7520F4</td>
<td>70</td>
<td>MOV SUB_DIV. #244, GET VALUE FOR NEXT ONE SECOND DELAY AND</td>
</tr>
<tr>
<td>0057 7520F4</td>
<td>71</td>
<td>GO THROUGH ELECTRICAL SYSTEM TEST CODE</td>
</tr>
<tr>
<td>005A 4390EO</td>
<td>72</td>
<td>ORL P1. #11100000B, SET CONTROL OUTPUTS HIGH</td>
</tr>
<tr>
<td>005D 43A007</td>
<td>73</td>
<td>ORL P2. #0000011B</td>
</tr>
<tr>
<td>0060 C295</td>
<td>74</td>
<td>CLR L.FRNT, FLOAT DRIVE COLLECTOR</td>
</tr>
<tr>
<td>0062 208428</td>
<td>75</td>
<td>JB TO_FAULT, TO SHOULD BE PULLED LOW</td>
</tr>
<tr>
<td>0065 D295</td>
<td>76</td>
<td>SETB L.FRNT, PULL COLLECTOR BACK DOWN</td>
</tr>
<tr>
<td>0067 C297</td>
<td>77</td>
<td>CLR L.DASH, REPEAT SEQUENCE FOR L.DASH.</td>
</tr>
<tr>
<td>0069 208421</td>
<td>78</td>
<td>JB TO_FAULT</td>
</tr>
<tr>
<td>006C D297</td>
<td>79</td>
<td>SETB L.DASH</td>
</tr>
<tr>
<td>006E C2A1</td>
<td>80</td>
<td>CLR L.REAR</td>
</tr>
<tr>
<td>0070 20841A</td>
<td>81</td>
<td>JB TO_FAULT</td>
</tr>
<tr>
<td>0073 D2A1</td>
<td>82</td>
<td>SETB L.REAR</td>
</tr>
<tr>
<td>0075 C296</td>
<td>83</td>
<td>CLR R.FRNT, R.FRNT.</td>
</tr>
<tr>
<td>0077 208413</td>
<td>84</td>
<td>JB TO_FAULT</td>
</tr>
<tr>
<td>007A D296</td>
<td>85</td>
<td>SETB R.FRNT</td>
</tr>
<tr>
<td>007C C2A0</td>
<td>86</td>
<td>CLR R.DASH, R.DASH.</td>
</tr>
<tr>
<td>007E 20840C</td>
<td>87</td>
<td>JB TO_FAULT</td>
</tr>
<tr>
<td>0081 D2A0</td>
<td>88</td>
<td>SETB R.DASH, AND R.REAR</td>
</tr>
<tr>
<td>0083 C2A2</td>
<td>89</td>
<td>CLR R.REAR</td>
</tr>
<tr>
<td>0085 208405</td>
<td>90</td>
<td>JB TO_FAULT</td>
</tr>
<tr>
<td>0088 D2A2</td>
<td>91</td>
<td>SETB R.REAR</td>
</tr>
<tr>
<td>92</td>
<td></td>
<td>WITH ALL COLLECTORS GROUNDED, TO SHOULD BE HIGH</td>
</tr>
<tr>
<td>94</td>
<td></td>
<td>IF SO, CONTINUE WITH INTERRUPT ROUTINE</td>
</tr>
<tr>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>008A 208402</td>
<td>96</td>
<td>JB TO_TOOSERV</td>
</tr>
<tr>
<td>008D B2A3</td>
<td>97</td>
<td>FAULT. CPL S_FAIL, ELECTRICAL FAILURE PROCESSING ROUTINE</td>
</tr>
<tr>
<td>98</td>
<td></td>
<td>(TOGGLE INDICATOR ONCE PER SECOND)</td>
</tr>
</tbody>
</table>

203830-27
LOC OBJ LINE SOURCE

100 CONTINUE WITH INTERRUPT PROCESSING
101
102 1) COMPUTE LOW BULB INTENSITY WHEN PARKING LIGHTS ARE ON
103
104 TOSERV MOV C_SUB_DIV_1 , START WITH 50 PERCENT.
105 009F A201 106 ANL C_SUB_DIV_0 , MASK DOWN TO 25 PERCENT.
107 0091 B200 108 ORL C_SUB_DIV_2 , BUILD BACK TO 62.5 PERCENT.
108 0093 B202 109 ANL C_PARK , GATE WITH PARKING LIGHT SWITCH.
109 0095 B205 110 MOV DIM_C , AND SAVE IN TEMP. VARIABLE
110
111 2) COMPUTE AND OUTPUT LEFT-HAND DASHBOARD INDICATOR
112
113 MOV C_L_TURN , SET CARRY IF TURN
114 0099 A203 115 ORL C_EMERG , OR EMERGENCY SELECTED.
115 0098 7201 116 ANL C_LO_FREQ , IF SO, GATE IN 1 Hz SIGNAL
116 009D B207 117 MOV L_DASH_C , AND OUTPUT TO DASHBOARD.
117
118 3) COMPUTE AND OUTPUT LEFT-HAND FRONT TURN SIGNAL
119
120 MOV F0_C , SAVE FUNCTION SO FAR
120 00A1 9205 121 ORL C_DIM , ADD IN PARKING LIGHT FUNCTION
121 00A3 7201 122 MOV L_FRNT_C , AND OUTPUT TO TURN SIGNAL
122 00A5 9205 123 ORL C_DIM , ADD IN PARKING LIGHT FUNCTION
123
124 4) COMPUTE AND OUTPUT LEFT-HAND REAR TURN SIGNAL
125
126 MOV C_BRAKE , GATE BRAKE PEDAL SWITCH
126 00A7 A200 127 ANL C_L_TURN , WITH TURN LEVER
127 00A9 B203 128 ORL C_F0 , INCLUDE TEMP. VARIABLE FROM DASH
128 00AB 7205 129 ORL C_DIM , AND PARKING LIGHT FUNCTION
129 00AD 7201 130 MOV L_REAR_C , AND OUTPUT TO TURN SIGNAL.
130 00AF 9201 131 ORL C_DIM , ADD IN PARKING LIGHT FUNCTION
131
132 5) REPEAT ALL OF ABOVE FOR RIGHT-HAND COUNTERPARTS.
133
134 MOV C_R_TURN , SET CARRY IF TURN
134 00B1 A204 135 ORL C_EMERG , OR EMERGENCY SELECTED.
135 00B3 7201 136 ANL C_LO_FREQ , IF SO, GATE IN 1 Hz SIGNAL
136 00B5 B207 137 MOV R_DASH_C , AND OUTPUT TO DASHBOARD.
137 00B7 9205 138 MOV F0_C , SAVE FUNCTION SO FAR.
138 00B9 9205 139 ORL C_DIM , ADD IN PARKING LIGHT FUNCTION
139 00BB 7201 140 MOV R_FRNT_C , AND OUTPUT TO TURN SIGNAL.
140 00BD 9205 141 MOV C_BRAKE , GATE BRAKE PEDAL SWITCH
141 00BF A200 142 ORL C_F0 , INCLUDE TEMP. VARIABLE FROM DASH
142 00C1 B203 143 ORL C_DIM , AND PARKING LIGHT FUNCTION
143 00C3 7205 144 MOV R_REAR_C , AND OUTPUT TO TURN SIGNAL.
144
145 RESTORE STATUS REGISTER AND RETURN
145
146
147 POP PSW , RESTORE PSW
147 00C9 D200 148 RETI , AND RETURN FROM INTERRUPT ROUTINE
148 00CB 32
## XREF SYMBOL TABLE LISTING

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE AND REFERENCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAKE N</td>
<td>BSEG</td>
<td>0090H 20# 125 140</td>
</tr>
<tr>
<td>DIM N</td>
<td>BSEG</td>
<td>00D1H 45# 108 120 128 136 143</td>
</tr>
<tr>
<td>EA N</td>
<td>BSEG</td>
<td>00AFH 6#</td>
</tr>
<tr>
<td>EMERG N</td>
<td>BSEG</td>
<td>0091H 21# 113 134</td>
</tr>
<tr>
<td>ETO N</td>
<td>BSEG</td>
<td>00A9H 63</td>
</tr>
<tr>
<td>FO N</td>
<td>BSEG</td>
<td>00D5H 119 127 137 142</td>
</tr>
<tr>
<td>FAULT L</td>
<td>CSEG</td>
<td>00BDH 75 78 81 84 87 90 97#</td>
</tr>
<tr>
<td>HI_FREQ N</td>
<td>BSEG</td>
<td>0000H 42#</td>
</tr>
<tr>
<td>INIT L</td>
<td>CSEG</td>
<td>0040H 50 58#</td>
</tr>
<tr>
<td>L_DASH N</td>
<td>BSEG</td>
<td>0097H 32# 77 79 115</td>
</tr>
<tr>
<td>L_FRNT N</td>
<td>BSEG</td>
<td>0095H 30# 74 76 121</td>
</tr>
<tr>
<td>L_REAR N</td>
<td>BSEG</td>
<td>00A1H 34# 80 82 129</td>
</tr>
<tr>
<td>L_TURN N</td>
<td>BSEG</td>
<td>0093H 23# 112 126</td>
</tr>
<tr>
<td>LO_FREQ N</td>
<td>BSEG</td>
<td>0007H 43# 114 135</td>
</tr>
<tr>
<td>P1 N</td>
<td>BSEG</td>
<td>0040H 20 21 22 23 24 30 31 32 72</td>
</tr>
<tr>
<td>P2 N</td>
<td>BSEG</td>
<td>00A0H 33 34 35 37 73</td>
</tr>
<tr>
<td>PARK N</td>
<td>BSEG</td>
<td>0092H 22# 107</td>
</tr>
<tr>
<td>PSW N</td>
<td>BSEG</td>
<td>00D0H 45 54 14B</td>
</tr>
<tr>
<td>R_DASH N</td>
<td>BSEG</td>
<td>00A0H 33# 86 88 136</td>
</tr>
<tr>
<td>R_FRNT N</td>
<td>BSEG</td>
<td>0096H 31# 83 85 139</td>
</tr>
<tr>
<td>R_REAR N</td>
<td>BSEG</td>
<td>00A2H 35# 89 91 144</td>
</tr>
<tr>
<td>R_TURN N</td>
<td>BSEG</td>
<td>0094H 24# 133 141</td>
</tr>
<tr>
<td>S_FAIL N</td>
<td>BSEG</td>
<td>00A3H 37# 97</td>
</tr>
<tr>
<td>SUB_DIV N</td>
<td>DSEG</td>
<td>0200H 41# 42 43 62 69 70 104 105 106</td>
</tr>
<tr>
<td>TO N</td>
<td>BSEG</td>
<td>0064H 75 78 81 84 87 90 96</td>
</tr>
<tr>
<td>TOSERV L</td>
<td>CSEG</td>
<td>00BFH 69 96 104#</td>
</tr>
<tr>
<td>THO N</td>
<td>BSEG</td>
<td>008CH 53 59</td>
</tr>
<tr>
<td>TLO N</td>
<td>BSEG</td>
<td>008AH 50</td>
</tr>
<tr>
<td>TMOD N</td>
<td>BSEG</td>
<td>0098H 60</td>
</tr>
<tr>
<td>TRO N</td>
<td>BSEG</td>
<td>008CH 65</td>
</tr>
<tr>
<td>UPDATE L</td>
<td>CSEG</td>
<td>0054H 55 69#</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETE. NO ERRORS FOUND
8051 Based CRT Terminal Controller
1.0 INTRODUCTION

This is the third application note that Intel has produced on CRT terminal controllers. The first Ap Note (ref. 1), written in 1977, used the 8080 as the CPU and required 41 packages including 11 LSI devices. In 1979, another application note (ref. 2) using the 8085 as the controller was produced and the chip count decreased to 20 with 11 LSI devices.

Advancing technology has integrated a complete system onto a single device that contains a CPU, program memory, data memory, serial communication, interrupt controller, and I/O. These "computer-on-a-chip" devices are known as microcontrollers. Intel's MCS®-51 microcontroller was chosen for this application because of its highly integrated functions. This CRT terminal design uses 12 packages with only 4 LSI devices.

This application note has been divided into five general sections:

1) CRT Terminal Basics
2) 8051 Description
3) 8276 Description
4) Design Background
5) System Description

2.0 CRT TERMINAL BASICS

A terminal provides a means for humans to communicate with a computer. Terminals may be as simple as a LED display and a couple of push buttons, or it may be an elaborate graphics system that contains a full function keyboard with user programmable keys, color CRT and several processors controlling its functions. This application note describes a basic low cost terminal containing a black and white CRT display, full function keyboard and a serial interface.

2.1 CRT Description

A raster scan CRT displays its images by generating a series of lines (raster) across the face of the tube. The electron beam usually starts at the top left hand corner moves left to right, back to the left of the screen, moves down one row and continues on to the right. This is repeated until the lower right hand of the screen is reached. Then the beam returns to the top left hand corner and refreshes the screen. The beam forms a zigzag pattern as shown in Figure 2.1.0.

Two independent operating circuits control this movement across the screen. The horizontal oscillator controls the left to right motion of the beam while the vertical controls the top to bottom movement. The vertical oscillator also tells the beam when to return to the upper left hand corner or "home" position.

As the electron beam moves across the screen under the control of the horizontal oscillator, a third circuit controls the current entering the electron gun. By varying the current, the image may be made as bright or as dim as the user desires. This control is also used to turn the beam off or "blank the screen".

When the beam reaches the right hand side of the screen, the beam is blanked so it does not appear on the screen as it returns to the left side. This "retrace" of the beam is at a much faster rate than it traveled across the screen to generate the image.

The time it takes to scan the whole screen and return to the home position is referred to as a "frame". In the United States, commercial television broadcast uses a horizontal sweep frequency of 15,750Hz which calculates out to 63.5 microseconds per line. The frame time is equal to 16.67 milliseconds or 60Hz vertical sweep frequency.

Although this is the commercial standard, many CRT displays operate from 18KHz to 30KHz horizontal frequency. As the horizontal frequency increases, the number of lines per frame increases. This increase in lines or resolution is needed for graphic displays and on special text editors that display many more lines of text than the standard 24 or 25 character lines.

Since the United States operates on a 60Hz A.C. power line frequency, most CRT monitors use 60Hz as the vertical frequency. The use of 60Hz as the vertical frequency allows the magnetic and electrical variations that can modulate the electron beam to be synchronized with the display, thus they go unnoticed. If a frequency other than 60Hz is used, special shielding and power supply regu-
lating is usually required. Very few CRTs operate on a vertical frequency other than 60Hz due to the increase in the overall system cost.

The CRT controller must generate the pulses that define the horizontal and vertical timings. On most raster scan CRTs the horizontal frequency may vary as much as 500Hz without any noticeable effect on the quality of the display. This variation can change the number of horizontal lines from 256 to 270 per frame.

The CRT controller must also shift out the information to be displayed serially to the circuit that controls the electron beam's intensity as it scans across the screen. The circuits that control the timing associated with the shifting of the information are known as the dot clock and the character clock. The character clock frequency is equal to the dot clock frequency divided by the number of dots it takes to form a character in the horizontal axis. The dot clock frequency is calculated by the following equation:

\[
\text{Dot Clock (Hz)} = (N + R) \times D \times L \times F
\]

where
- \( N \) is the number of displayed characters per row,
- \( R \) is the number of character times for the retrace,
- \( D \) is the number of dots per character in the horizontal axis,
- \( L \) is the number of horizontal lines per frame,
- \( F \) is the frame rate in Hz.

In this design \( N = 80, R = 20, D = 7, L = 270 \), and \( F = 60\text{Hz} \). Plugging in the numbers results in a dot clock frequency of 11.34MHz.

The retrace number may vary on each design because it is used to set the left and right hand margins on the CRT. The number of dots per character is chosen by the designer to meet the system needs. In this design, a 5 x 7 dot matrix and 2 blank dots between each character (see Figure 2.1.1) makes \( D = 5 + 2 = 7 \).

The following equation can be used to figure the number of lines per frame:

\[
L = (H \times Z) + V
\]

where
- \( H \) is the number of horizontal lines per character,
- \( Z \) is the number of character lines per frame,
- \( V \) is the number of horizontal line times during the vertical retrace

In this design \( H \) is equal to the 7 horizontal dots per character plus 3 blank dots between each row which adds up to 10. Also 25 lines of characters are displayed, so \( Z = 25 \). The vertical retrace time is variable to set the top and bottom margins on the CRT and in this design is equal to 20. Plugging in the numbers gives \( L = 270 \) lines per frame.

### 2.2 Keyboard

A keyboard is the common way a human enters commands and data to a computer. A keyboard consists of a matrix of switches that are scanned every couple of milliseconds by a keyboard controller to determine if one of the keys has been pressed. Since the keyboard is made up of mechanical switches that tend to bounce or "make and break" contact everytime they are pressed, debouncing of the switches must also be a function of the keyboard controller. There are dedicated keyboard controllers available that do everything from scanning the keyboard, debouncing the keys, decoding the ASCII code for that key closure to flagging the CPU that a valid key has been depressed. The keyboard controller may present the information to the CPU in parallel form or in a serial data stream.

This Application Note integrates the function of the keyboard controller into the 8051 which is also the terminal controller. Provisions have been made to interface the 8051 to a keyboard that uses a dedicated keyboard controller. The 8051 can accept data from the keyboard controller in either parallel or serial format.

### 2.3 Serial Communications

Communication between a host computer and the CRT terminal can be in either parallel or serial data format. Parallel data transmission is needed in high end graphic terminals where great amounts of information must be transferred.

One can rarely type faster than 120 words per minute, which corresponds to 12 characters per second or 1 character per 83 milliseconds. The utilization of a parallel port cannot justify the cost associated with the drivers and the amount of wire needed to perform this transmission. Full duplex serial data transmission requires 3 wires and two
drivers to implement the communication channel between the host computer and the terminal. The data rate can be as high as 19200 BAUD in the asynchronous serial format. BAUD rate is the number of bits per second received or transmitted. In the asynchronous serial format, 10 bits of information is required to transmit one character. One character per 500 microseconds or 1,920 characters per second would then be transmitted using 19.2 KBAUD.

This application note uses the 8051 serial port configured for full duplex asynchronous serial data transmission. The software for the 8051 has been written to support variable BAUD rates from 150 BAUD up to 9.6 KBAUD.

3.0 **8051 DESCRIPTION**

The 8051 is a single chip high-performance microcontroller. A block diagram is shown in figure 3.0.0. The 8051 combines CPU; Boolean processor; 4K x 8 ROM; 128 x 8 RAM; 32 I/O lines; two 16-bit timer/event counters; a five-source, two-priority-level, nested interrupt structure; serial I/O port for either multiprocessor communications, I/O expansion, or full duplex UART; and on-chip oscillator and clock circuits.

3.1 **CPU**

Efficient use of program memory results from an instruction set consisting of 49 single-byte, 45 two-byte and 17 three-byte instructions. Most arithmetic, logical and branching operations can be performed using an instruction that appends either a short address or a long address. For example, branches may use either an offset that is relative to the program counter which takes two bytes or a direct 16-bit address which takes three bytes to perform. As a result, 64 instructions operate in one machine cycle, 45 in two machine cycles, and the multiply and divide instruction execute in 4 machine cycles.

The 8051 has five addressing modes for source operands: Register, Direct, Register-Indirect, Immediate, and Based-Register-plus Index-Register-Indirect Addressing.

The Boolean Processor can be thought of as a separate one-bit CPU. It has its own accumulator (the carry bit), instruction set for data moves, logic, and control transfer, and its own bit addressable RAM and I/O. The bit-manipulating instructions provide optimum code and speed efficiency for handling on chip peripherals. The
Boolean processor also provides a straightforward means of converting logic equations directly into software. Complex combinational logic functions can be resolved without extensive data movement, byte masking, and test-and-branch trees.

3.2 On-Chip Ram

The CPU manipulates operands in four memory spaces. These are the 64K-byte Program Memory, 64K-byte External Data Memory, 128-byte Internal Data Memory, and 128-byte Special Function Registers (SFRs). Four Register Banks (each with 8 registers), 128 addressable bits, and the Stack reside in the internal Data RAM. The Stack size is limited only by the available Internal Data RAM and its location is determined by the 8-bit Stack Pointer. All registers except for the Program Counter and the four 8-Register Banks reside in the SFR address space. These memory mapped registers include arithmetic registers, pointers, I/O ports, and registers for the interrupt system, timers, and serial channel.

Registers in the four 8-Register Banks can be addressed by Register, Direct, or Register-Indirect Addressing modes. The 128 bytes of internal Data Memory can be addressed by Direct or Register-Indirect modes while the SFRs are only addressed directly.

3.3 I/O Ports

The 8051 has instructions that can treat the 32 I/O lines as 32 individually addressable bits or as 4 parallel 8-bit ports addressable as Ports 0, 1, 2, and 3.

Resetting the 8051 writes a logical 1 to each pin on port 0 which places the output drivers into a high-impedance mode. Writing a logical 0 to a pin forces the pin to ground and sinks current. Re-writing the pin high will place the pin in either an open drain output or high-impedance input mode.

Ports 1, 2, and 3 are known as quasi-bidirectional I/O pins. Resetting the device writes a logical one to each pin. Writing a logical 0 to the pin will force the pin to ground and sink current. Re-writing the pin high will place the pin in an output mode with a weak depletion pullup FET or in the input mode. The weak pullup FET is easily overcome by a TTL output.

Ports 0 and 2 can also be used for off-chip peripheral expansion. Port 0 provides a multiplexed low-order address and data bus while Port 2 contains the high-order address when using external Program Memory or more than 256 byte external Data Memory.

Port 3 pins can also be used to provide external interrupt request inputs, event counter inputs, the serial port TXD and RXD pins and to generate control signals used for writing and reading external peripherals.

3.4 Interrupt System

External events and the real-time-driven on-chip peripherals require service by the CPU asynchronous to the execution of any particular section of code. A five-source, two-level, nested interrupt system ties the real time events to the normal program execution.

The 8051 has two external interrupt sources, one interrupt from each of the two timer/counters, and an interrupt from the serial port. Each interrupt vectors the program execution to its own unique memory location for servicing the interrupt. In addition, each of the five sources can be individually enabled or disabled as well as assigned to one of the two interrupt priority levels available on the 8051.

Up to two additional external interrupts can be created by configuring a timer/counter to the event counter mode. In this mode the timer/counter increments on command by either the T0 or T1 pin. An interrupt is generated when the timer/counter overflows. Thus if the timer/counter is loaded with the maximum count, the next high-to-low transition of the event counter input will cause an interrupt to be generated.

3.5 Serial Port

The 8051's serial port is useful for linking peripheral devices as well as multiple 8051s through standard asynchronous protocols with full duplex operation. The serial port also has a synchronous mode for expansion of I/O lines using shift registers. This hardware serial port saves ROM code and permits a much higher transmission rate than could be achieved through software. The processor merely needs to read or write the serial buffer in response to an interrupt. The receiver is double buffered to eliminate the possibility of overrun if the processor failed to read the buffer before the beginning of the next frame.

The full duplex asynchronous serial port provides the means of communication with standard UART devices such as CRT terminals and printers.

The reader should refer to the microcontroller handbook for a complete discussion of the 8051 and its various modes of operation.

4.0 8276 Description

The 8276's block diagram and pin configuration are shown in Figure 4.0.0. The following sections describe the general capabilities of the 8276.
4.1 CRT Display Refreshing

The 8276, having been programmed by the system designer for a specific screen format, generates a series of Buffer Ready signals. A row of characters is then transferred by the system controller from the display memory to the 8276’s row buffers. The row buffers are filled by deselecting the 8276 CS and asserting the BS and WR signals. The 8276 presents the character codes to an external character generator ROM by using outputs CC0–CC6. The parallel data from the outputs of the character generator is converted to serial information that is clocked by external dot timing logic into the video input of the CRT.

The character rows are displayed on the CRT one line at a time. Line count outputs LC0–LC3 select the current line information from the character generator ROM. The display process is illustrated in Figure 4.1.0. This process is repeated for each display character row. At the beginning of the last display row the 8276 generates an interrupt request by raising its INT output line. The interrupt request is used by the 8051 system controller to reinitialize its load buffer pointers for the next display refresh cycle.

Proper CRT refreshing requires that certain 8276 parameters be programmed at system initialization time. The 8276 has two types of internal registers; the write only Command (CREG) and Parameter (PREG) Registers, and the read only Status Register (SREG). The 8276 expects to receive a command followed by 0 to 4 parameter bytes depending on the command. A summary of the 8276’s instruction set is shown in Figure 4.1.1.

The 8276 allows the designer flexibility in the display format. The display may be from 1 to 80 characters per row, 1 to 64 rows per screen, and 1 to 16 horizontal lines per character row. In addition, four cursor formats are available; blinking, non-blinking, underline, and reverse video. The cursor position is programmable to anywhere on the screen via the Load Curser command.
4.2 CRT Timing

The 8276 provides two timing outputs for controlling the CRT. The Horizontal Retrace Timing and Control (HRTC) and Vertical Retrace Timing and Control (VRTC) signals are used for synchronizing the CRT horizontal and vertical oscillators. A third output, VSP (Video Suppress), provides a signal to the dot timing logic to blank the video signal during the horizontal and vertical retraces. LTEN (Light Enable) is used to provide the ability to force the video output high regardless of the state of the VSP signal. This feature is used to place the cursor on the screen and to control attribute functions.

RVV (Reverse Video) output, if enabled, will cause the system to invert its video output. The fifth timing signal output, HLGT (highlight) allows the flexibility to increase the CRT beam intensity to a greater than normal level.

Figure 4.1.0 8276 Row Display
**4.3 Special Functions**

### 4.3.1 Special Codes

The 8276 recognizes four special codes that may be used to reduce memory, software, or system controller overhead. These characters are placed within the display memory by the system controller. The 8276 performs certain tasks when these codes are received in its row buffer memory.

1) **End of Row Code** — Activates VSP. VSP remains active until the end of the line is reached. While VSP is active the screen is blanked.

2) **End Of Row-Stop Buffer Loading Code** — Causes the Buffer Ready control logic to stop requesting buffer transfers for the rest of the row. It affects the display the same as End of Row Code.

3) **End Of Screen Code** — Activates VSP. VSP remains active until the end of the frame is reached.

4) **End Of Screen-Stop Buffer Loading Code** — Causes the Buffer Ready control logic to stop requesting buffer transfers until the end of the frame is reached. It affects the display the same way as the End of Screen code.

### 4.3.4 Programmable Buffer Loading Control

The 8276 can be programmed to request 1, 2, 4, or 8 characters per Buffer load. The interval between loads is also programmable. This allows the designer the flexibility to tailor the buffer transfer overhead to fit the system needs.

Each scan line requires 63.5 microseconds. A character line consists of 10 scan lines and takes 635 microseconds to form. The 8276 row buffer must be filled within the 635 microseconds or an under run condition will occur within the 8276 causing the screen to be blanked until the next vertical retrace. This blanking will be seen as a flicker in the display.

### 5.0 DESIGN BACKGROUND

A fully functional, microcontroller-based CRT terminal was designed and constructed using the 8051 and the 8276. The terminal has many of the functions that are found in commercially available low cost terminals. Sophisticated features such as programmable keys can be added easily with modest amounts of software.

The 8051’s functions in this application note include: up to 9.6K BAUD full duplex serial transmission; decoding special messages sent from the host computer; scanning, debouncing, and decoding a full function keyboard; writing to the 8276 row buffer from the display RAM without the need for a DMA controller; and scrolling the display.

The 8276 CRT controller’s functions include: presenting the data to the character generator; providing the timing signals needed for horizontal and vertical retrace; and providing blanking and video information.

### 5.1 Design Philosophy

Since the device count relates to costs, size, and reliability of a system, arriving at a minimum device count without degrading the performance was a driving force for this application note. LSI devices were used where possible to maintain a low chip count and to make the design cycle as short as possible.

PL/M-51 was chosen to generate the majority of the software for this application because it models the human thought process more closely than assembly language. Consequently it is easier and faster to write programs using PL/M-51 and the code is more likely to be correct because less chance exists to introduce errors.
PL/M-51 programs are easier to read and follow than assembly language programs, and thus are easier to modify and customize to the end user's application. PL/M-51 also offers lower development and maintenance costs than assembly language programming.

PL/M-51 does have a few drawbacks. It is not as efficient in code generation relative to assembly language and thus may also run slower.

This application note uses the 8051's interrupts to control the servicing of the various peripherals. The speed of the main program is less critical if interrupts are used. In the last two application notes on terminal controllers, a criterion of the system was the time required for receiving an incoming serial byte, decoding it, performing the function requested, scanning the keyboard, debouncing the keys, and transmitting the decoded ASCII code must be less than the vertical refresh time. Using the 8051 and its interrupts makes this time constraint irrelevant.

5.2 System Target Specifications

The design specifications for the CRT terminal design is as follows:

Display Format
- 80 characters/display row
- 25 display lines

Character Format
- 5 x 7 character contained within a 7 x 10 frame
- First and seventh columns blanked
- Ninth line cursor position
- Programmable delay blinking underline cursor

Control Characters Recognized
- Backspace
- Linefeed
- Carriage Return
- Form Feed

Escape Sequences Recognized
- ESC A, Curser up
- ESC B, Curser down
- ESC C, Curser right
- ESC D, Curser left
- ESC E, Clear screen
- ESC F, Move addressable curser
- ESC H, Home curser
- ESC J, Erase from curser to the end the screen
- ESC K, Erase the current line

Characters Displayed
- 96 ASCII Alphanumeric Characters

Characters Transmitted
- 96 ASCII Alphanumeric Characters
- ASCII Control Character Set
- ASCII Escape Sequence Set
- Auto Repeat

Display Memory
- 2K x 8 static RAM

Data Rate
- Variable rate from 150 to 9600 BAUD

CRT Monitor
- Ball Bros TV-12, 12MHZ Black and White

Keyboard
- Any standard undecoded keyboard (2 key lock-out)
- Any standard decoded keyboard with output enable pin
- Any standard decoded serial keyboard up to 150 BAUD

Scrolling Capability
Compatible With Wordstar

6.0 SYSTEM DESCRIPTION

A block diagram of the CRT terminal is shown in figure 6.0.0. The diagram shows only the essential system features. A detailed schematic of the CRT terminal is contained in the Appendix 7.1.

The "brains" of the CRT terminal is the 8051 microcontroller. The 8276 is the CRT controller in the system, and a 2716 EPROM is used as the character generator. To handle the high speed portion of the CRT, the 8276 is surrounded by a handful of TTL devices. A 2K x 8 static RAM was used as the display memory.

Following the system reset, the 8276 is initialized for curser type, number of characters per line, number of lines, and character size. The display RAM is initialized to all "spaces" (ASCII 20H). The 8051 then writes the "start display" command to the 8276. The local/line input is sampled to determine the terminal mode. If the terminal is on-line, the BAUD rate switches are read and the serial port is set up for full duplex UART mode. The processor then is put into a loop waiting to service the serial port fifo or the 8276.

The serial port is programmed to have the highest priority interrupt. If the serial port generates an interrupt, the processor reads the buffer, puts the character in a generated fifo that resides in the 8051's internal RAM, increments the fifo pointer, sets the serial interrupt flag and returns.
The main program determines if it is a displayable character, a Control word or an ESC sequence and either puts the character in the display buffer or executes the appropriate command sent from the host computer.

If the 8276 needs servicing, the 8051 fills the row buffer for the CRT display’s next line. If the 8276 generates a vertical retrace interrupt, the buffer pointers are reloaded with the display memory location that corresponds to the first character of the first display line on the CRT. The vertical retrace also signals the processor to read the keyboard for a key closure.

### 6.1 Hardware Description

The following section describes the unique characteristics of this design.

#### 6.1.1 Peripheral Address Map

The display RAM, 8276 registers, and the 8276 row buffers are memory mapped into the external data RAM address area. The addresses are as follows:

- **Read and Write External Display RAM** — Address 1000H to 17CFH
- **Write to 8276 row buffers from Display RAM** — Address 1800H to 1FCFH
- **Write to 8276 Command Register (CREG)** — Address 0001H
- **Write to 8276 Parameter Register (PREG)** — Address 0000H
- **Read from 8276 Status Register (SREG)** — Address 0001H

Three general cases can be explored; reading and writing the display RAM, writing to the 8276 row buffers, and reading and writing the 8276’s control registers.

As mentioned previously the 8051 fills the 8276 row buffer without the need of a DMA controller. This is accomplished by using a Quad 2-input multiplexor (Figure 6.1.0) as the transfer logic shown in the block diagram. The address line, P2.3, is used to select either of the two inputs. When the address line is low the RD and WR lines perform their normal functions, that is read and write the
8276 or the external display RAM depending on the states of their respective chip selects. If the address line is high, the 8051 RD line is transformed into BS and WR signals for the 8276. While holding the address line high, the 8051 executes an external data move (MOVX) from the display RAM to the accumulator which causes the display RAM to output the addressed byte onto the data bus. Since the multiplexor turns the same 8051 RD pulses into BS and WR pulses to the 8276, the data bus is thus read into the 8276 as a Buffer transfer. This scheme allows 80 characters to be transferred from the display RAM into the 8276 within the required character line time of 635 microseconds. The 8051 easily meets this requirement by accomplishing the task within 350 microseconds.

6.1.2 Scanning The Keyboard

Throughout this project, provision have been made to make the overall system flexible. The software has been written for various keyboards and the user simply needs to link different program modules together to suit their needs.

6.1.2.1 Undecoded Keyboard

Incorporating an undecoded keyboard controller into the other functions of the 8051 shows the flexibility and overall CPU power that is available. The keyboard in this case is a full function, non-buffered 8 x 8 matrix of switches for a total of 64 possible keys. The 8 send lines are connected to a 3-to-8 open-collector decoder as shown in Figure 6.1.1. Three high order address lines from the 8051 are the decoder inputs. The enabling of the decoder is accomplished through the use of the PSEN signal from the 8051 which makes the architecture of the separate address space for the program memory and the external data RAM work for us to eliminate the need to decode addresses externally. The move code (MOVC) instruction allows each scan line of the keyboard to be read with one instruction.

The keyboard is read by bringing one of the eight scan lines low sequentially while reading the return lines which are pulled high by an external resistor. If a switch is

![Figure 6.1.1 Keyboard](2-86)
4.3 Special Functions

4.3.1 Special Codes

The 8276 recognizes four special codes that may be used to reduce memory, software, or system controller overhead. These characters are placed within the display memory by the system controller. The 8276 performs certain tasks when these codes are received in its row buffer memory.

1) End of Row Code — Activates VSP. VSP remains active until the end of the line is reached. While VSP is active the screen is blanked.

2) End Of Row-Stop Buffer Loading Code — Causes the Buffer Ready control logic to stop requesting buffer transfers for the rest of the row. It affects the display the same as End of Row Code.

3) End Of Screen Code — Activates VSP. VSP remains active until the end of the frame is reached.

4) End Of Screen-Stop Buffer Loading Code — Causes the Buffer Ready control logic to stop requesting buffer transfers until the end of the frame is reached. It affects the display the same way as the End of Screen code.

4.3.4 Programmable Buffer Loading Control

The 8276 can be programmed to request 1, 2, 4, or 8 characters per Buffer load. The interval between loads is also programmable. This allows the designer the flexibility to tailor the buffer transfer overhead to fit the system needs.

Each scan line requires 63.5 microseconds. A character line consists of 10 scan lines and takes 635 microseconds to form. The 8276 row buffer must be filled within the 635 microseconds or an under run condition will occur within the 8276 causing the screen to be blanked until the next vertical retrace. This blanking will be seen as a flicker in the display.

5.0 DESIGN BACKGROUND

A fully functional, microcontroller-based CRT terminal was designed and constructed using the 8051 and the 8276. The terminal has many of the functions that are found in commercially available low cost terminals. Sophisticated features such as programmable keys can be added easily with modest amounts of software.

The 8051’s functions in this application note include: up to 9.6K BAUD full duplex serial transmission; decoding special messages sent from the host computer; scanning, debouncing, and decoding a full function keyboard; writing to the 8276 row buffer from the display RAM without the need for a DMA controller; and scrolling the display.

The 8276 CRT controller’s functions include: presenting the data to the character generator; providing the timing signals needed for horizontal and vertical retrace; and providing blanking and video information.

5.1 Design Philosophy

Since the device count relates to costs, size, and reliability of a system, arriving at a minimum device count without degrading the performance was a driving force for this application note. LSI devices were used where possible to maintain a low chip count and to make the design cycle as short as possible.

PL/M-51 was chosen to generate the majority of the software for this application because it models the human thought process more closely than assembly language. Consequently it is easier and faster to write programs using PL/M-51 and the code is more likely to be correct because less chance exists to introduce errors.
PUM-51 programs are easier to read and follow than assembly language programs, and thus are easier to modify and customize to the end user's application. PUM-51 also offers lower development and maintenance costs than assembly language programming.

PUM-51 does have a few drawbacks. It is not as efficient in code generation relative to assembly language and thus may also run slower.

This application note uses the 8051's interrupts to control the servicing of the various peripherals. The speed of the main program is less critical if interrupts are used. In the last two application notes on terminal controllers, a criterion of the system was the time required for receiving an incoming serial byte, decoding it, performing the function requested, scanning the keyboard, debouncing the keys, and transmitting the decoded ASCII code must be less than the vertical refresh time. Using the 8051 and its interrupts makes this time constraint irrelevant.

### 5.2 System Target Specifications

The design specifications for the CRT terminal design is as follows:

**Display Format**
- 80 characters/display row
- 25 display lines

**Character Format**
- 5 x 7 character contained within a 7 x 10 frame
- First and seventh columns blanked
- Ninth line cursor position
- Programmable delay blinking underline cursor

**Control Characters Recognized**
- Backspace
- Linefeed
- Carriage Return
- Form Feed

**Escape Sequences Recognized**
- ESC A, Cursor up
- ESC B, Cursor down
- ESC C, Cursor right
- ESC D, Cursor left
- ESC E, Clear screen
- ESC F, Move addressable cursor
- ESC H, Home cursor
- ESC J, Erase from cursor to the end the screen
- ESC K, Erase the current line

**Characters Displayed**
- 96 ASCII Alphanumeric Characters

**Characters Transmitted**
- 96 ASCII Alphanumeric Characters
- ASCII Control Character Set
- ASCII Escape Sequence Set
- Auto Repeat

**Display Memory**
- 2K x 8 static RAM

**Data Rate**
- Variable rate from 150 to 9600 BAUD

**CRT Monitor**
- Ball Bros TV-12, 12MHZ Black and White

**Keyboard**
- Any standard undecoded keyboard (2 key lock-out)
- Any standard decoded keyboard with output enable pin
- Any standard decoded serial keyboard up to 150 BAUD

**Scrolling Capability**

**Compatible With Wordstar**

### 6.0 SYSTEM DESCRIPTION

A block diagram of the CRT terminal is shown in figure 6.0.0. The diagram shows only the essential system features. A detailed schematic of the CRT terminal is contained in the Appendix 7.1.

The "brains" of the CRT terminal is the 8051 microcontroller. The 8276 is the CRT controller in the system, and a 2716 EPROM is used as the character generator. To handle the high speed portion of the CRT, the 8276 is surrounded by a handful of TTL devices. A 2K x 8 static RAM was used as the display memory.

Following the system reset, the 8276 is initialized for cursor type, number of characters per line, number of lines, and character size. The display RAM is initialized to all "spaces" (ASCII 20H). The 8051 then writes the "start display" command to the 8276. The local/line input is sampled to determine the terminal mode. If the terminal is on-line, the BAUD rate switches are read and the serial port is set up for full duplex UART mode. The processor then is put into a loop waiting to service the serial port fifo or the 8276.

The serial port is programmed to have the highest priority interrupt. If the serial port generates an interrupt, the processor reads the buffer, puts the character in a generated fifo that resides in the 8051's internal RAM, increments the fifo pointer, sets the serial interrupt flag and returns.
The main program determines if it is a displayable character, a Control word or an ESC sequence and either puts the character in the display buffer or executes the appropriate command sent from the host computer.

If the 8276 needs servicing, the 8051 fills the row buffer for the CRT display's next line. If the 8276 generates a vertical retrace interrupt, the buffer pointers are reloaded with the display memory location that corresponds to the first character of the first display line on the CRT. The vertical retrace also signals the processor to read the keyboard for a key closure.

Three general cases can be explored: reading and writing the display RAM, writing to the 8276 row buffers, and reading and writing the 8276's control registers.

As mentioned previously the 8051 fills the 8276 row buffer without the need of a DMA controller. This is accomplished by using a Quad 2-input multiplexor (Figure 6.1.0) as the transfer logic shown in the block diagram. The address line, P2.3, is used to select either of the two inputs. When the address line is low the RD and WR lines perform their normal functions, that is read and write the

### 6.1 Hardware Description

The following section describes the unique characteristics of this design.

#### 6.1.1 Peripheral Address Map

The display RAM, 8276 registers, and the 8276 row buffers are memory mapped into the external data RAM address area. The addresses are as follows:

- **Read and Write External Display RAM** — Address 1000H to 17CFH
- **Write to 8276 row buffers from Display RAM** — Address 1800H to 1FCFH
- **Write to 8276 Command Register (CREG)** — Address 0001H
- **Write to 8276 Parameter Register (PREG)** — Address 0000H
- **Read from 8276 Status Register (SREG)** — Address 0001H

---

![CRT Terminal Controller Block Diagram](image)

**Figure 6.0.0** CRT Terminal Controller Block Diagram

![Simplified Version Of The Transfer Logic](image)

**Figure 6.1.0** Simplified Version Of The Transfer Logic
8276 or the external display RAM depending on the states of their respective chip selects. If the address line is high, the 8051 RD line is transformed into BS and WR signals for the 8276. While holding the address line high, the 8051 executes an external data move (MOVX) from the display RAM to the accumulator which causes the display RAM to output the addressed byte onto the data bus. Since the multiplexor turns the same 8051 RD pulses into BS and WR pulses to the 8276, the data bus is thus read into the 8276 as a Buffer transfer. This scheme allows 80 characters to be transferred from the display RAM into the 8276 within the required character line time of 635 microseconds. The 8051 easily meets this requirement by accomplishing the task within 350 microseconds.

6.1.2 Scanning The Keyboard

Throughout this project, provision have been made to make the overall system flexible. The software has been written for various keyboards and the user simply needs to link different program modules together to suit their needs.

6.1.2.1 Undecoded Keyboard

Incorporating an undecoded keyboard controller into the other functions of the 8051 shows the flexibility and over all CPU power that is available. The keyboard in this case is a full function, non-buffered 8 x 8 matrix of switches for a total of 64 possible keys. The 8 send lines are connected to a 3-to-8 open-collector decoder as shown in Figure 6.1.1. Three high order address lines from the 8051 are the decoder inputs. The enabling of the decoder is accomplished through the use of the PSEN signal from the 8051 which makes the architecture of the separate address space for the program memory and the external data RAM work for us to eliminate the need to decode addresses externally. The move code (MOVC) instruction allows each scan line of the keyboard to be read with one instruction.

The keyboard is read by bringing one of the eight scan lines low sequentially while reading the return lines which are pulled high by an external resistor. If a switch is
closed, the data bus line is connected through the switch to the low output of the decoder and one of the data bus lines will be read as a 0. By knowing which scan line detected a key closure and which data bus line was low, the ASCII code for that key can easily be looked up in a matrix of constants. PL/M-51 has the ability to handle arrays and structured arrays, which makes the decoding of the keyboard a trivial task.

Since the Shift, Cap Lock, and Control keys may change the ASCII code for a particular key closure, it is essential to know the status of these pins while decoding the keyboard. The Shift, Cap Lock, and Control keys are therefore not scanned but are connected to the 8051 port pins where they can be tested for closure directly.

The 8 receive lines are connected to the data bus through germanium diodes which chosen for their low forward voltage drop. The diodes keep the keyboard from interfering with the data bus during the times the keyboard is not being read. The circuit consisting of the 3-to-8 decoder and the diodes also offers some protection to the 8051 from possible Electrostatic Discharge (ESD) damage that could be transmitted through the keyboard.

6.1.2.2 Decoded Keyboard

A decoded keyboard can easily be connected to the system as shown in Figure 6.1.2. Reading the keyboard can be evoked either by interrupts or by software polling.

The software to periodically read a decoded keyboard was not written for this application note but can be accomplished with one or two PL/M-51 statements in the READER routine.

A much more interesting approach would be to have the servicing of the keyboard be interrupt driven. An additional external interrupt is created by configuring timer/counter 0 into an event counter. The event counter is initialized with the maximum count. The keyboard controller would inform the 8051 that a valid key has been depressed by pulling the input pin T0 low. This would overflow the event counter, thus causing an interrupt. The interrupt routine would simply use a MOVC (PSEN is connected to the output enable pin of the keyboard controller) to read the contents of the keyboard controller onto the data bus, reinitialize the counter to the maximum count and return from the interrupt.

6.1.2.3 Serial Decoded Keyboard

The use of detachable keyboards has become popular among the manufacturers of keyboards and personal computers. This terminal has provisions to use such a keyboard.

The keyboard controller would scan the keyboard, de-bounce the key and send back the ASCII code for that key closure. The message would be in an asynchronous serial format.

The flowchart for a software serial port is shown in Figure 6.1.3. An additional external interrupt is created as discussed for the decoded keyboard but the use in this case would be to detect a start bit. Once the beginning of the start bit has been detected, the timer/counter 0 is configured to become a timer. The timer is initialized to cause an interrupt one-half bit time after the beginning of the start bit. This is to validate the start bit. Once the start bit is validated, the timer is initialized with a value to cause an interrupt one bit time later to read the first data bit. This process of interrupting to read a data bit is repeated until all eight data bits have been received. After all 8 data bits are read, the software serial port is read once more to detect if a stop bit is present. If the stop bit is not present, an error flag is set, all pointers and flags are reset to their initial values, and the timer/counter is re-configured to an event counter to detect the next start bit. If the stop bit is present, a valid flag is set and the flags and counter are reset as previously discussed.
6.1.4 System Timings

The requirements for the BALL BROTHERS. TV-12 monitor's operation is shown in table 6.1.0. From the monitor's parameters, the 8276 specifications and the system target specifications the system timing is easily calculated.

The 8276 allows the vertical retrace to be only an integer multiple of the horizontal character lines. Twenty-five display lines and a character frame of $7 \times 10$ are required from the target specification which will require 250 horizontal lines. If the horizontal frequency is to be within 500 Hz of 15,750 Hz, we must choose either one or two character line times for horizontal retrace. To allow for a little more margin at the top and bottom of the screen, two character line times was chosen for the vertical retrace. This choice yields $250 + 20 = 270$ total character lines per frame. Assuming 60 Hz vertical retrace frequency:

$$60 \text{ Hz} \times 270 = 16,200 \text{ Hz}$$

and

$$1/16,200 \text{ Hz} \times 20 \text{ horizontal sync times} = 1.2345 \text{ milliseconds}$$
Table 6.1.0  CRT Monitor's Operational Requirements

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical Blanking Time</td>
<td>800 μsec nominal</td>
</tr>
<tr>
<td>(VRTC)</td>
<td></td>
</tr>
<tr>
<td>Vertical Drive Pulsewidth</td>
<td>300 μsec ≤ PW ≤ 1.4 ms</td>
</tr>
<tr>
<td>Horizontal Blanking Time</td>
<td>11 μsec nominal</td>
</tr>
<tr>
<td>(HRTC)</td>
<td></td>
</tr>
<tr>
<td>Horizontal Drive Pulsewidth</td>
<td>25 μsec ≤ PW ≤ 30 μsec</td>
</tr>
<tr>
<td>Horizontal Repetition Rate</td>
<td>15,750 ± 500 pps</td>
</tr>
</tbody>
</table>

The 1.2345 milliseconds of retrace time meets the nominal VRTC and vertical drive pulse width time of .3mSec to 1.4mSec for the Ball monitor.

The next parameter to find is the horizontal retrace time which is wholly dependent on the monitor used. Usually it lies between 15 and 30 percent of the total horizontal line time.

Since most designs display a fixed number of characters per line it is useful to express the horizontal retrace time as a given number of character times. In this design, 80 characters are displayed, and it was experimentally found that 20 character times for the horizontal retrace gave the best results. It should be noted if too much time was given for retrace, there would be less time to display the characters and the display would not fill out the screen. Conversely, if not enough time is given for retrace, the characters would seem to run off the screen.

One hundred character times per complete horizontal line means that each character needs:

\[
\frac{1/(16,200 \text{ Hz})}{100 \text{ character times}} = 617.3 \text{ nanoseconds}
\]

If we multiply the 20 character times needed to retrace by 617.3 nanoseconds needed for each character, we find 12.345 microseconds are allocated for retrace. This value falls short of the 25 to 30 microseconds required by the horizontal drive of the Ball monitor. To correct for this, a 74LS123 one-shot was used to extend the horizontal drive pulse width.

The dot clock frequency is easy to calculate now that we know the horizontal frequency. Since each character is formed by seven dots in the horizontal axis, the dot clock period would be the character clock (617.3 nanoseconds) divided by the 7 which is equal to 11.34 MHz. The basic dot timing and CRT timing are shown in the Appendix.

6.2  Software Description

6.2.1  Software Overview

The software for this application was written in a "foreground-background" format. The background programs are all interrupt driven and are written in assembly language due to time constraints. The foreground programs are for the most part written in PL/M-51 to ease the programming effort. A number of subroutines are written in assembly language due to time constraints during execution. Subroutines such as clearing display lines, clearing the screen, and scanning the keyboard require a great deal of 16 bit adds and compares and would execute much slower and would require more code space if written in PL/M-51. The background and foreground programs talk to each other through a set of flags. For example, the PL/M-51 foreground program tests "SERIAL$INT" to determine if a serial port interrupt had occurred and a character is waiting to be processed.

6.2.2  The Background Program

Two interrupt driven routines, VERT and BUFFER, (see Fig. 6.2.0) request service every 16.67 milliseconds and 617 microseconds respectively. VERT's request comes during the last character row of the display screen. This routine resets the buffer pointers to the first CRT display line in the display memory. VERT is also used as a time base for the foreground program. VERT sets the flag, SCAN, to tell the foreground program (PL/M-51) that it is time to scan the Keyboard. VERT also increments a counter used for the delay between transmitting characters in the AUTOSREPEAT routine.

The BUFFER routine is executed once per character row. BUFFER uses the multiplexor discussed earlier to fill the 8276's row buffer by executing 80 external data moves and incrementing the Data Pointer between each move.
The MOVX reads the display RAM and writes the character into the row buffer during the same instruction.

SERBUF is an interrupt driven routine that is executed each time a character is received or transmitted through the on-chip serial port. The routine first checks if the interrupt was caused by the transmit side of the serial port, signaling that the transmitter is ready to accept another character. If the transmitter caused the interrupt, the flag "TRANSMITSINT" is set which is checked by the foreground program before putting a character in the buffer for transmission.

If the receiver caused the interrupt, the input buffer on the serial port is read and fed into the fifo that has been manufactured in the internal RAM and increments the fifo pointer "FIFO." The flag "SERIALSINT" is then set, telling the foreground program that there is a character in the fifo to be processed. If the read character is an ESC character, the flag "ESCEQ" is set to tell the foreground program that an escape sequence is in the process of being received.

### 6.2.3 The Foreground Program

The foreground program is documented in the Appendix. The foreground program starts off by initializing the 8276 as discussed earlier. After all variables and flags are initialized, the processor is put into a loop waiting for either VERT to set SCAN so the program can scan the keyboard, or for the serial port to set SERIALSINT so the program can process the incoming character.

The vertical retrace is used to time the delay between keyboard scans. When VERT gets set, the assembly language routine READER is called. READER scans the keyboard, writing each scan into RAM to be processed later. READER controls two flags, KEY0 and SAME. KEY0 is set when all 8 scans determine that no key is pressed. SAME is set when the same key that was pressed last time the keyboard was read is still pressed.

After READER returns execution to the main program, the flags are tested. If the KEY0 flag is set the main program goes back to the loop waiting for the vertical retrace or a serial port interrupt to occur. If the SAME flag is set the main program knows that the closed key has been debounced and decoded so it sends the already known ASCII code to the AUTO$REPEAT routine which determines if that character should be transmitted or not.

If KEY0 and SAME are not set, signifying that a key is pressed but it is not the same key as before, the foreground program determines if the results from the scan are valid. First all eight scans are checked to see if only one key was closed. If only one key is closed, the ASCII code is determined, modified if necessary by the Shift, Cap Lock, or Control keys. The NEWSKEY and VALID flags are then set. The next time READER is called, if the same key is still pressed, the SAME flag will be set, causing the AUTO$REPEAT subroutine to be called as just discussed. Since the keyboard is read during the vertical retrace, 16.67 milliseconds has elapsed between the detection of the pressed key and revalidating that the key is still pressed before transmitting it, thus effectively debouncing the key.

The AUTO$REPEAT routine is written to transmit any key that the NEWSKEY flag is set for. The counter that is incremented each time the vertical refresh interrupt is serviced causes a programmable delay between the first transmission and subsequent auto repeat transmission. Once the NEWSKEY character is sent, the counter is initialized. Each time the AUTO$REPEAT routine is called, the counter is checked. Only when the counter overflows will the next character be transmitted. After the initial delay, a character will be transmitted every other time the routine is called as long as the key remains pressed.

### 6.2.3.1 Handling Incoming Serial Data

One of the criteria for this application note was to make the software less time dependent. By creating a fifo to store incoming characters until the 8051 has time to pro-
cess them, software timing becomes less critical. This application note uses up to 8 levels of the fifo at 9.2KBAUD, and 1 level at 4.8KBAUD and lower. As discussed earlier, the interrupt service routine for the serial port uses the fifo to store incoming data, increments the fifo pointer, “FIFO”, and sets SERIALSINT to tell the main program that the fifo needs servicing. Once the main program detects that SERIALSINT is set the routine DECIPHER is executed.

DECIPHER has three separate blocks; a block for decoding displayable characters, a block for processing Escape sequences, and a block for processing Control codes. Each block works on the fifo independently. Before exiting a block, the contents of the fifo are shifted up by the amount of characters that were processed in that particular block. The shifting of the characters insures that the beginning of the fifo contains the next character to be processed. FIFO is then decremented by the number of characters processed.

Let’s look at this process more closely. Figure 6.2.1-A shows a representation of a fifo containing 5 characters. The first three characters in the fifo contain displayable characters, A, B, and C respectively with the last two characters being an ESC sequence for moving the curser up one line (ESC A) and FIFO points to the next available location to be filled by the serial port interrupt routine, in this case, 5.

When DECIPHER is executed, the first block begins looking at the first character of the fifo for a displayable character. If the character is displayable, it is placed into the display RAM and the software pointer “TOP” that points to the character that is being processed is incremented to the next character. The character is then looked at to see if it too is displayable and if it is, it’s placed in the display RAM. The process of checking for displayable characters is continued until either the fifo is empty or a non-displayable character is detected. In our example, three characters are placed into the display RAM before a non-displayable character is detected. At this point the fifo looks like figure 6.2.1-B.

Before entering the next block, the remaining contents of the fifo between TOP, that is now pointing to 1BH and (FIFO-1) are moved up in the fifo by the amount of characters processed, in this example three. TOP is reset to 0 and FIFO is decremented by 3. The serial port interrupt is inhibited during the time the contents of the fifo and the pointers are being manipulated. The fifo now looks like figure 6.2.1-C.

The execution is now passed to the next block that processes ESC sequences. The first location of the fifo is examined to see if it is an ESC character (1BH). If not, the execution is passed to the next block of DECIPHER that processes Control codes. In this case the fifo does contain an ESC code. The flag ESCSEQ is checked to see if the 8051 is in the process of receiving an ESC sequence thus signifying that the next byte of the sequence has not been received yet. If the ESCSEQ is not set, the next character in the fifo is checked for a valid escape code and the proper subroutine is then called. The fifo contents are then shifted as discussed for the previous block. Due to the length of time that is needed to execute an ESC code sequence or a Control code, only one ESC code and/or Control code can be processed each time DECIPHER is executed.

If at the end of the DECIPHER routine, FIFO contains a 0, the flag SERSINT is reset. If SERSINT remains set, DECIPHER will be executed immediately after returning to the main program if SCAN had not been set during the execution of the DECIPHER routine, otherwise DECIPHER will be called after the keyboard is read.

### 6.2.4 Memory Pointers and Scrolling

The curser always points to the next location in display memory to be filled. Each time a character is placed in the display memory, the curser position needs to be tested to determine if the curser should be incremented to the beginning of the next line of the display or simply moved to the next position on the current display line. The curser position pointers are then updated in both the 8276 and the internal registers in the 8051.
When the 2000th character is entered into the display memory, a full display page has been reached signaling the need for the display to scroll. The memory pointer that points to the display memory that contains the first character of the first display line, LINE0, prior to scrolling contains 1800H which is the starting address of the display memory. Each scrolling operation adds 80 (50H) to LINE0 which will now point to the following row in memory as shown in figure 6.2.2-B. LINE0 is used during the vertical refresh routine to re-initialize the pointers associated with filling the 8276 row buffers.

The display memory locations that were the first line of the CRT display now becomes the last line of the CRT display. Incoming characters are now entered into the display memory starting with 1800H, which is now the first character of the last line of the display screen.

---

**Figure 6.2.2 Pointer Manipulation During Scrolling**

2-92
6.2.5 Software Timing

The use of interrupts to tie the operation of the foreground program to the real-time events of the background program has made the software timing non-critical for this system.

6.3 System Operation

Following the system reset, the 8051 initializes all on-chip peripherals along with the 8276 and display ram. After initialization, the processor waits until the fifo has a character to process or is flagged that it is time to scan the keyboard. This foreground program is interrupted once every 617 microseconds to service the 8276 row buffers. The 8051 is also interrupted each 16.67 milliseconds to re-initialize LINE0 and to flag the foreground program to read the keyboard.

As discussed earlier, a special technique of rapidly moving the contents of the display RAM to the 8276 row buffers without the need of a DMA device was employed. The characters are then synchronously transferred to the character generator via CC0-CC6 and LC0-LC2 which are used to display one line at a time. Following the transfer of the first line to the dot timing logic, the line count is incremented and the second line is selected. This process continues until the last line of the character is transferred.

The dot timing logic latches the output of the character ROM in a parallel in, serial out synchronous shift register. The shift register's output constitutes the video information to the CRT.
Appendix 7.1 CRT Schematics

[Diagram of CRT Schematics]
Appendix 7.2 Dot Timing

Character Counter State

745163 Counter Outputs

Character Clock

8276 Character Output (CGO-CC6)

Shift Register Output (74186)

First Character Video Out

Second Character Video Out

11.34 MHz

7404

10pF

DATA BUS

5V

A1 B1 Q1 CEXT

1/2 74123

Character Clock

LCO-6C2 COO-CCE

Character Generator

8 10 2 7 6

7410

74175

745163

VSP (8275)

LTEN (8275)

HRTC (8275)

VRTC (8275)

VSP Delayed

LTEN Delayed

HRTC Delayed

VRTC Delayed

CRT Monitor

+V

+V

+V

+V

1K

1K

1K

1K

Horizontal Drive

Vertical Drive

HORIZONTAL DRIVE

VERTICAL DRIVE

11.34 MHz Dot Clock

BC DEFGH

CLK 74106 QO SHIFTL/OAD

SHIFT REGISTER

LOAD

11.34 MHz Dot Clock

7410
Appendix 7.3  CRT System Timing
## Appendix 7.4 Escape/Control/Display Character Summary

<table>
<thead>
<tr>
<th>BIT</th>
<th>CONTROL CHARACTERS</th>
<th>DISPLAYABLE CHARACTER</th>
<th>ESCAPE SEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>NUL @</td>
<td>SP</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>DLE 0</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>SOH A</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>STX B</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>EOT D</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>ENQ E</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>ACK F</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>BEL G</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>BS H</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>HT I</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>LF J</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>VT K</td>
<td>ESC</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>FF L</td>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>CR M</td>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>SO N</td>
<td>RS</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>SI S</td>
<td>US</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table Notes:
- **Shaded blocks** — functions terminal will react to. Others can be generated but are ignored upon receipt.
- **Blank spaces** — do not react to, are ignored.
Appendix 7.5 Character Generator

As previously mentioned, the character generator used in this terminal is a 2716 EPROM. A 1K by 8 device would have been sufficient since a 128 character 5 by 7 dot matrix only requires 8K of memory. A custom character set could have been stored in the second 1K bytes of the 2716. Any of the free I/O pins on the 8051 could have been used to switch between the character sets.

The three low-order line count outputs (LCO–LC2) from the 8276 are connected to the three low-order address lines of the character generator. The CC0–CC6 output lines are connected to the A3–A9 lines of the character generator.

The output of the character generator is loaded into the shift register. The serial output of the shift register is the video output to the CRT.

Let's assume that the letter “E” is to be displayed. The ASCII code for “E” (45H) is presented to the address lines A2–A9 of the character generator. The scan lines (LCO–LC2) will now count from 0 to seven to form the character as shown in Figure 7.5.0. The same procedure is used to form all 128 possible characters. For reference Appendix 7.6 contains the HEX dump of the character generator used in this terminal.

<table>
<thead>
<tr>
<th>Rom Address</th>
<th>Rom Hex Output</th>
<th>Bit Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>228H</td>
<td>3E</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>229H</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>22AH</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>22BH</td>
<td>0E</td>
<td></td>
</tr>
<tr>
<td>22CH</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>22DH</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>22EH</td>
<td>3E</td>
<td></td>
</tr>
<tr>
<td>22FH</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

Bits 0, 6 and 7 are not used.

*note bit output is backward from convention.

Figure 7.5.0 Character Generator
Appendix 7.7 Composite Video

In this design it was assumed that the CRT monitor required a separate horizontal drive, vertical drive, and video input. Many monitors require a composite video signal. The schematic shown in Figure 7.7.0 illustrate how to generate a composite video from the output of the 8276.

The dual one-shots are used to provide a small delay and the proper horizontal and vertical pulse to the composite video monitor. The delay introduced in the horizontal and vertical timing is used to center the display. The 7486 is used to mix the vertical and horizontal retrace. Q1 mix the video and retrace signals along with providing the proper D.C. levels.

Figure 7.7.0 Composite Video
Appendix 7.8 Software Documentation

MEMORY MAP ASSOCIATED WITH PERIPHERAL DEVICES (USING MOVX):

- 8051 WR AND READ DISPLAY RAM - ADDRESS 1000H TO 17FFH
- 8051 WR DISPLAY RAM TO THE 8276 - ADDRESS 1800H TO 1FCH
- 8276 COMMAND ADDRESS - ADDRESS 0001H
- 8276 PARAMETER ADDRESS - ADDRESS 0000H
- 8276 STATUS REGISTER - ADDRESS 0001H

MEMORY MAP FOR READING THE KEYBOARD (USING MOVX):

KEYBOARD ADDRESS - ADDRESS 10FFH TO 17FFH

/* BEGIN BY PUTTING THE ASCII CODE FOR BLANK IN THE DISPLAY RAM */

INIT:
{FILL 2000 LOCATIONS IN THE DISPLAY RAM WITH SPACES (ASCII 20H)}

/* INITIALIZE POINTERS, RAM BITS, ETC. */

- INITIALIZE POINTERS AND FLAGS
- INITIALIZE TOP OF THE CRT DISPLAY "LINE0"=1800H
- INITIALIZE 8276 RASTER POINTER "RASTER" =1800H
- INITIALIZE DISPLAY RAM POINTER=0000H

/* INITIALIZE THE 8276 */

- RESET THE 8276
- INITIALIZE 8276 TO 80 CHARACTER/ROW
- INITIALIZE 8276 TO 25 ROWS PER FRAME
- INITIALIZE 8276 TO 10 LINES PER ROW
- INITIALIZE 8276 TO NON-BLINKING UNDERLINE CURSOR
- INITIALIZE CURSOR TO HOME POSITION (00,00) (UPPER LEFT HAND CORNER)
- START DISPLAY
- [ENABLE 8276 INTERRUPT]

/* SET UP 8051 INTERRUPTS AND PRIORITIES */

- SERIAL PORT HAS HIGHEST INTERRUPT PRIORITY
- [DISABLE EXTERNAL INTERRUPTS]
/*PROCEDURE SCANNER: THIS PROCEDURE SCANS THE KEYBOARD AND DETERMINES IF A SINGLE VALID KEY HAS BEEN PUSHED. IF TRUE THEN THE ASCII EQUIVALENT WILL BE TRANSMITTED TO THE HOST COMPUTER.*/

SCANNER:
{ENABLE 8051 GLOBAL INTERRUPT BIT}

/* PROGRAMMABLE DELAY FOR THE CURSOR BLINK */

IF {30 VERTICAL RETRACE INTERRUPTS HAVE OCCURRED (CURSER$COUNT=1FH)} THEN DO;
    [COMPLEMENT CURSER$COUNT]
    [CLEAR CURSER$COUNT]
    IF {CURSER IS TO BE OFF (CURSER$ON=0)} THEN [MOVE CURSOR OFF THE SCREEN]
    CALL LOAD$CURSER;
END;

IF {THE LOCAL LINE SWITCH HAS CHANGED STATE} THEN DO;
    IF {IN LOCAL MODE} THEN [DISABLE SERIAL PORT INTERRUPT]
    ELSE CALL CHECK$BAUD$RATE;
END;

DO WHILE {IN BETWEEN VERTICAL REFRESHES}:
    IF {THE FIFO HAS A CHARACTER TO PROCESS (SERIAL$INT=1)} THEN CALL DECIPHER;
END;

CALL READER;

IF {THE PRESENT PRESSED KEY IS EQUAL TO THE LAST KEY PRESSED AND VALID=1} THEN CALL AUTO$REPEAT;
ELSE DO;
    IF {A KEY IS PRESSED BUT NOT THE SAME ONE AS THE LAST KEYBOARD SCAN} THEN DO;
        IF {ONLY ONE KEY IS PRESSED} THEN
            [GET THE ASCII CODE FOR IT]
            [SET NEW$KEY AND VALID FLAGS]
        ELSE [RESET VALID AND NEW$KEY FLAGS]
    END;
    ELSE [THE KEYBOARD MUST NOT HAVE A KEY PRESSED SO RESET VALID$KEY AND NEW$KEY FLAGS]
END;
GOTO SCANNER;

/*PROCEDURE AUTO$REPEAT: THIS PROCEDURE WILL PERFORM AN AUTO REPEAT FUNCTION BY TRANSMITTING A CHARACTER EVERY OTHER TIME THIS ROUTINE IS CALLED. THE AUTO REPEAT FUNCTION IS ACTIVATED AFTER A FIXED DELAY PERIOD AFTER THE FIRST CHARACTER IS SENT*/

AUTO$REPEAT:
IF {THE KEY PRESSED IS NEW (NEW$KEY=1)} THEN DO;
    [CLEAR THE DIVIDE BY TWO COUNTER "TRANSMIT$TOGGLE"]
    [INITIALIZE THE DELAY COUNTER "TRANSMIT$COUNT" TO $00H]
    CALL TRANSMIT;
    [CLEAR NEW$KEY]
    /* FIRST CHARACTER */
END;
ELSE
DO;
    IF [TRANSMIT$COUNT IS NOT EQUAL TO 0] THEN
        DO;
            [INCREMENT TRANSMIT$COUNT]
            IF TRANSMIT$COUNT = OFF THEN
                DO;
                    CALL TRANSMIT;
                    [CLEAR TRANSMIT$COUNT]
                END;
            END;
        END;
    ELSE
        DO;
            [TURN THE CURSOR ON DURING THE AUTO REPEAT FUNCTION]
            IF TRANSMIT$TOGGLE = 1 THEN
                CALL TRANSMIT;
                [COMPLEMENT TRANSMIT$TOGGLE]
            END;
        END;
    END AUTO$REPEAT;
END;

/* PROCEDURE TRANSMIT- ONCE THE HOST COMPUTER SIGNALS THE 8051H BY BRINGING THE CLEAR-TO-SEND LINE LOW, THE ASCII CHARACTER IS PUT INTO THE SERIAL PORT.*/

TRANSMIT;
PROCEDURE;
IF [THE TERMINAL IS ON-LINE] THEN
    DO;
        [WAIT UNTIL THE CLEAR$TO$SEND LINE IS LOW AND UNTIL THE 8051 SERIAL PORT TX IS NOT BUSY (TRANSMIT$INT=1)]
        [TRANSMIT THE ASCII CODE]
        [CLEAR THE FLAG "TRANSMIT$INT", THE SERIAL PORT SERVICE ROUTINE WILL SET THE FLAG WHEN THE SERIAL PORT IS FINISHED TRANSMITTING]
    END;
ELSE [THE TERMINAL IS IN THE LOCAL MODE]
    DO;
        [PUT THE ASCII CODE IN THE FIFO]
        [INCREMENT THE FIFO POINTER]
        [SET SERIAL$INT]
    END;
END TRANSMIT;
/*  PROCEDURE DECRYPT: THIS PROCEDURE DECODES THE HOST COMPUTER'S MESSAGES AND DETERMINES WHETHER IT IS A DISPLAYABLE CHARACTER, CONTROL SEQUENCE, OR AN ESCAPE SEQUENCE. THE PROCEDURE THEN ACTS ACCORDINGLY. */

DECRYPT:
START$DECRYPT:

VALID$RECEPTION=0;
DO WHILE (THE FIFO IS NOT EMPTY AND THE CHARACTER IS DISPLAYABLE)
    RECEIVE={ASCII CODE}
    CALL DISPLAY;
    [NEXT CHARACTER]
END;

IF [CHARACTERS WERE DISPLAYED] THEN
    [DISABLE SERIAL PORT INTERRUPT]
    [MOVE THE REMAINING CONTENTS OF THE FIFO UP TO THE BEGINNING OF THE FIFO]
    [SET THE VALID$RECEPTION FLAG]

IF [THE FIFO IS EMPTY] THEN [CLEAR THE "SERIAL$INT FLAG AND RETURN]

IF [THE NEXT CHARACTER IS AN "ESC" CODE ] THEN
    DO:
        [LOOK AT THE CHARACTER IN THE FIFO AFTER THE ESC CODE AND CALL THE CORRECT SUBROUTINE]
        /* ESC A */
        CALL UP$CURSOR;
        /* ESC B */
        CALL DOWN$CURSOR;
        /* ESC C */
        CALL RIGHT$CURSOR;
        /* ESC D */
        CALL LEFT$CURSOR;
        /* ESC E */
        CALL CLEAR$SCREEN;
        /* ESC F */
        CALL MOVE$CURSOR;
        /* ESC H */
        CALL HOME;
        /* ESC J */
        CALL ERASE$FROM$CURSOR$TO$END$OF$SCREEN;
        /* ESC K */
        CALL BLINE;

    [DISABLE THE SERIAL PORT INTERRUPT]
    [MOVE THE REMAINING CONTENTS OF THE FIFO UP TO THE BEGINNING OF THE FIFO]
    [SET THE "VALID$RECEPTION" FLAG]

    IF [THE FIFO IS EMPTY] THEN [CLEAR THE SERIAL$INT FLAG AND RETURN]
END;

2-105
IF (THE NEXT CHARACTER IS A CONTROL CODE) THEN
DO;
[CALL THE RIGHT SUBROUTINE]
CALL LEFT$CURSER; /* CTL H */
CALL LINES$FEED; /* CTL J */
CALL CLEAR$SCREEN; /* CTL L */
CALL CARRIAGE$RETURN; /* CTL M */
[DISABLE THE SERIAL PORT INTERRUPT]
[MOVE THE REMAINING CONTENTS OF THE FIFO UP TO THE BEGINNING OF THE FIFO]
[ENABLE THE SERIAL PORT INTERRUPT]
[SET THE "VALID$RECEPTION" FLAG]
END;
IF (NO VALID CODE WAS RECEIVED ("VALID$RECEPTION" IS 0)) THEN
[THROW THE CHARACTER OUT AND MOVE THE REMAINING CONTENTS OF THE FIFO]
[UP TO THE BEGINNING]
IF (THE FIFO IS EMPTY) THEN [CLEAR THE SERIAL$INT FLAG AND RETURN]
END DECIPHER;

PROCEDURE DISPLAY: THIS PROCEDURE WILL TAKE THE BYTE IN RAM LABELED RECEIVE AND PUT IT INTO THE DISPLAY RAM. */

DISPLAY:
[PUT INTO THE DISPLAY RAM LOCATION POINTED TO BY "DISPLAY$RAM$POINTER THE CONTENTS OF RECEIVE]
IF (THE END OF THE DISPLAY MEMORY HAS BEEN REACHED) THEN
RESET "DISPLAY$RAM$POINTER" TO THE BEGINNING OF THE RAM
ELSE
INCREMENT "DISPLAY$RAM$POINTER"
IF (THE CURSOR IS IN THE LAST COLUMN OF THE CRT DISPLAY) THEN
DO;
MOVE THE CURSOR BACK TO THE BEGINNING OF THE LINE
IF (THE NEW DISPLAY RAM LOCATION HAS A END-OF-LINE CHARACTER IN IT) THEN
CALL FILL;
IF (THE CURSOR IS ON THE LAST LINE OF THE CRT DISPLAY) THEN
CALL SCROLL;
ELSE
MOVE THE CURSOR TO THE NEXT LINE
END;
ELSE
INCREMENT THE CURSOR TO THE NEXT LOCATION
[TURN THE CURSOR ON ]
CALL LOAD$CURSER;
END DISPLAY;
/*      PROCEDURE LINESPEED       */

LINESPEED:

IF (THE CURSOR IS IN THE LAST LINE OF THE CRT DISPLAY) THEN
  CALL SCROLL;
ELSE
  DO:
    {MOVE THE CURSOR TO THE NEXT LINE}
    {TURN THE CURSOR ON}
    CALL LOAD$CURSER;
  END;

IF (THE DISPLAY$RAM$POINTER IS ON THE LAST LINE IN THE DISPLAY RAM) THEN
  MOVE THE DISPLAY$RAM$POINTER TO THE FIRST LINE IN THE DISPLAY RAM
ELSE
  MOVE THE DISPLAY$RAM$POINTER TO THE NEXT LINE IN THE DISPLAY RAM

IF (THE FIRST CHARACTER IN THE NEW LINE CONTAINS AN END-OF-LINE CHARACTER) THEN
  CALL FILL;
END LINE$FEED;

/*      PROCEDURE SCROLL       */

SCROLL:

CALL BLANK;
{DISABLE VERTICAL RETRACE INTERRUPT}

IF (THE FIRST LINE OF THE CRT CONTAINS THE LAST LINE OF THE DISPLAY MEMORY) THEN
  MOVE THE POINTER "LINEO" TO THE BEGINNING OF THE DISPLAY MEMORY
ELSE
  MOVE "LINEO" TO THE NEXT LINE IN THE DISPLAY MEMORY
{ENABLE VERTICAL RETRACE INTERRUPT}
END SCROLL;

/*      PROCEDURE CLEAR SCREEN       */

CLEAR$SCREEN:

CALL HOME;
CALL ERASES$FROM$CURSER$TO$END$OF$SCREEN;
END CLEAR$SCREEN;
/*    PROCEDURE HOME: THIS PROCEDURE MOVES THE CURSOR TO THE 0,0 POSITION */

HOME:

[MOVE THE CURSOR POSITION TO THE UPPER LEFT HAND CORNER OF THE CRT]
[TURN THE CURSOR ON]
CALL LOAD$CURSOR;
[MOVE THE DISPLAY$RAM$POINTERS TO THE CORRECT LOCATION IN THE DISPLAY RAM]
END HOME;

/*    PROCEDURE ERASE FROM CURSOR TO END OF SCREEN: */

ERASE$FROM$CURSOR$TO$END$OF$SCREEN:

CALL BLINE;       /* ERASE CURRENT LINE */

IF [THE CURSOR IS NOT ON THE LAST LINE OF THE CRT DISPLAY] THEN
    [STARTING WITH THE NEXT LINE] PUT AN END-OF-LINE CHARACTER (0F1H)
    [IN THE DISPLAY RAM LOCATIONS THAT CORRESPOND TO THE BEGINNING OF
    THE CRT DISPLAY LINES UNTIL THE BOTTOM OF THE CRT SCREEN HAS BEEN REACHED]
END;

END ERASE$FROM$CURSOR$TO$END$OF$SCREEN;

/*    PROCEDURE MOV$CURSOR: THIS PROCEDURE IS USED IN CONJUNCTION WITH WORDSTAR
    IF A ESC F IS RECEIVED FROM THE HOST COMPUTER, THE TERMINAL CONTROLLER WILL
    READ THE NEXT TWO BYTES TO DETERMINE WHERE TO MOVE THE CURSOR. THE FIRST BYTE
    IS THE ROW INFORMATION FOLLOWED BY THE COLUMN INFORMATION */

MOV$CURSOR:

[WAIT UNTIL THE FIFO HAS RECEIVED THE NEXT TWO CHARACTERS]
[MOVE THE CURSOR TO THE LOCATION SPECIFIED IN THE ESCAPE SEQUENCE]
[MOVE THE DISPLAY$RAM$POINTERS TO THE CORRECT LOCATION]

IF THE FIRST CHARACTER IN THE NEW LINE HAS AN END-OF-LINE CHARACTER] THEN
    CALL FILL;
END;

[DISABLE THE SERIAL PORT INTERRUPT]
[MOVE THE REMAIN CONTENTS OF THE FIFO UP TWO LOCATIONS IN MEMORY]
[DECLINEMENT THE FIFO BY TWO]
[ENABLE THE SERIAL PORT INTERRUPT]

END MOV$CURSOR;

/*    PROCEDURE LEFT$CURSOR: THIS PROCEDURE MOVES THE CURSOR LEFT ONE COLUMN
    BY SUBTRACTING 1 OF THE CURSOR COLUMN RAM LOCATION THEN CALL LOAD CURSOR */

LEFT$CURSOR:

IF [THE CURSOR IS NOT IN THE FIRST LOCATION OF A LINE] THEN
    DO;
        [MOVE THE CURSOR LEFT BY ONE LOCATION]
        [TURN THE CURSOR ON]
        CALL LOAD$CURSOR;
        [DECREMENT THE DISPLAY$RAM$POINTERS BY ONE]
    END;

END LEFT$CURSOR;
/* PROCEDURE RIGHT CURSER: THIS PROCEDURE MOVES THE CURSOR RIGHT ONE COLUMN 
   BY ADDING 1 TO THE CURSOR COLUMN RAM LOCATION THEN CALL LOAD CURSER */

RIGHT$CURSER:
IF {THE CURSOR IS NOT IN THE LAST POSITION OF THE CRT LINE} THEN
   DO;
      [MOVE THE CURSOR RIGHT BY ONE LOCATION]
      [TURN THE CURSOR ON]
      CALL LOAD$CURSER;
      [INCREMENT THE DISPLAY$RAM$POINTER BY ONE]
   END;
END RIGHT$CURSER;

/* PROCEDURE UP CURSER: THIS PROCEDURE MOVES THE CURSOR UP ONE ROW 
   BY SUBTRACTING 1 TO THE CURSOR ROW RAM LOCATION THEN CALL LOAD CURSER */

UP$CURSER:
IF {THE CURSOR IS NOT ON THE FIRST LINE OF THE CRT DISPLAY} THEN
   DO;
      [MOVE THE CURSOR UP ONE LINE]
      [TURN ON THE CURSOR]
      CALL LOAD$CURSER;
      IF {THE DISPLAY$RAM$POINTER IS IN THE FIRST LINE OF DISPLAY MEMORY} THEN
         [MOVE THE DISPLAY$RAM$POINTER TO THE LAST LINE OF DISPLAY MEMORY]
      ELSE
         [MOVE THE DISPLAY$RAM$POINTER UP ONE LINE IN DISPLAY MEMORY]
      IF {THE FIRST LOCATION OF THE NEW LINE CONTAINS AN END-OF-LINE CHARACTER} THEN
         CALL FILL;
      END;
   END;
END UP$CURSER;

/* PROCEDURE DOWN CURSER: THIS PROCEDURE MOVES THE CURSOR DOWN ONE ROW 
   BY ADDING 1 TO THE CURSOR ROW RAM LOCATION THEN CALL LOAD CURSER */

DOWN$CURSER:
IF {THE CURSOR IS NOT ON THE LAST LINE OF THE CRT DISPLAY} THEN
   DO;
      [TURN THE CURSOR ON]
      [MOVE THE CURSOR TO THE NEXT LINE]
      CALL LOAD$CURSER;
      IF {THE DISPLAY$RAM$POINTER IS NOT ON THE LAST LINE OF THE DISPLAY MEMORY} THEN
         [MOVE THE DISPLAY$RAM$POINTER TO THE NEXT LINE IN THE DISPLAY MEMORY]
      ELSE
         [MOVE THE DISPLAY$RAM$POINTER TO THE FIRST LINE IN THE DISPLAY MEMORY]
      IF {THE FIRST CHARACTER IN THE NEW LINE IS AN END-OF-LINE CHARACTER} THEN
         CALL FILL;
      END;
   END;
END DOWN$CURSER;
PROCEDURE CARRIAGE$RETURN

CARRIAGE$RETURN:

[MOVE THE DISPLAY$RAM$POINTER TO THE BEGINNING OF THE CURRENT LINE IN THE DISPLAY MEMORY]
[MOVE THE CURSOR TO THE BEGINNING OF THE CURRENT LINE OF THE CRT DISPLAY]
[TURN THE CURSOR ON]
CALL LOAD$CURSER;
END CARRIAGE$RETURN;

PROCEDURE LOAD CURSER: LOAD CURSER TAKES THE VALUE HELD IN RAM AND LOADS IT INTO THE 8276 CURSER REGISTER. */

LOAD$CURSER:
PROCEDURE;
IF [THE CURSER IS ON] THEN
[MOVE THE CURSER BACK INTO THE CRT DISPLAY]
[DISABLE BUFFER INTERRUPT]
[WRITE TO THE 8276 CURSER REGISTERS THE X,Y LOCATIONS]
[ENABLE BUFFER INTERRUPT]

END LOAD$CURSER;

PROCEDURE CHECK BAUD RATE: THIS PROCEDURE READS THE THREE PORT PINS ON P1 AND SETS UP THE SERIAL PORT FOR THE SPECIFIED BAUD RATE */

CHECK$BAUD$RATE:

[SET TIMER 1 TO MODE 1 AND AUTO RELOAD]
[TURN TIMER ON]
[ENABLE SERIAL PORT INTERRUPT]
[READ BAUD RATE SWITCHES AND SET UP RELOAD VALUE]

; /* 00 IS NOT ALLOWED */
THl=040H; /* 150 BAUD */
THl=060H; /* 300 BAUD */
THl=080H; /* 600 BAUD */
THl=0B8H; /* 1200 BAUD */
THl=0F4H; /* 2400 BAUD */
THl=0FAH; /* 4800 BAUD */
THl=0FFH; /* 9600 BAUD */

END CHECK$BAUD$RATE;
PROCEDURE READER: THIS PROCEDURE IS WRITTEN IN ASSEMBLY LANGUAGE. THE EXTERNAL PROCEDURE SCANS THE 8 LINES OF THE KEYBOARD AND READS THE RETURN LINES. THE STATUS OF THE 8 RETURN LINES ARE THEN STORED IN INTERNAL MEMORY ARRAY CALLED CURRENTKEY */

READER:

[INITIALIZE FLAGS "KEY0"=0, "SAME"=1, 0 COUNTER=0]

DO UNTIL [ALL 8 KEYBOARD SCAN LINES ARE READ]

[READ KEYBOARD SCAN]

IF [NO KEY WAS PressED] THEN

[INCREMENT 0 COUNTER]

ELSE

IF [THE KEY PRESSED WAS NOT THE SAME KEY THAT WAS PRESSED THE LAST TIME THE KEYBOARD WAS READ] THEN

[CLEAR "SAME" AND WRITE NEW SCAN RESULT TO CURRENTKEY RAM ARRAY]

END;

IF [ALL 8 SCANS Didn'T HAVE A KEY PRESSED (0 COUNTER<>8)] THEN

[SET KEY0, AND CLEAR SAME]

END READER;

PROCEDURE BLANK: THIS EXTERNAL PROCEDURE FILLS LINE0 WITH SPACES (20H ASCII) DURING THE SCROLL ROUTINES. */

BLANK:

DO I= [BEGINNING OF THE CRT DISPLAY (LINE0)] TO [LINE0 + 50H]

[DISPLAY RAM POINTED TO BY "I" = SPACE (ASCII 20H)]

NEXT I

END;

END BLANK;

PROCEDURE BLINE: THIS EXTERNAL PROCEDURE BLANKS FROM THE CURSOR TO THE END OF THE DISPLAY LINE */

BLINE:

DO I= [CURRENT CURSOR POSITION ON CRT DISPLAY] TO [END OF ROW]

[DISPLAY RAM POINTED TO BY "I" = SPACE (ASCII 20H)]

NEXT I

END;

END BLINE;

PROCEDURE FILL: THIS EXTERNAL PROCEDURE FILLS A DISPLAY LINE WITH SPACES */

FILL:

DO I= [BEGINNING OF THE LINE THAT THE CURSOR IS ON] TO [END OF THE ROW]

[DISPLAY RAM POINTED TO BY "I" = SPACE (ASCII 20H)]

NEXT I

END;

END FILL;
Appendix 7.9 Software Listings

PL/M-51 COMPILER

ISIS-II PL/M-51 V1.1
COMPILER INVOKED BY: PLM51 :FL1:CRTPM.SRC

$OPTIMIZE(1)
$NOINTEGRATOR
$ROM(LARGE)

 /*****************************************************************************
 *****************************************************************************/

*****************************************************************************/

**********
**********
**********
**********
**********
**********
**********
**********

*****************************************************************************

MEMORY MAP ASSOCIATED WITH PERIPHERAL DEVICES (USING MOVM):

- 8051 WR AND READ DISPLAY RAM—ADDRESS 100H TO 17FH
- 8051 WR DISPLAY RAM TO THE 8276—ADDRESS 180H TO 1FFH
- 8276 COMMAND ADDRESS—ADDRESS 0001H
- 8276 PARAMETER ADDRESS—ADDRESS 0000H
- 8276 STATUS REGISTER—ADDRESS 0001H

MEMORY MAP FOR READING THE KEYBOARD (USING MOVC):

- KEYBOARD ADDRESS—ADDRESS 10FFH TO 17FFH

THE FOLLOWING SOFTWARE SWITCHES MUST BE SET ACCORDING TO THE TYPE OF
KEYBOARD THAT IS GOING TO BE USED.

SW1—SET WHEN USING AN UNDECODED KEYBOARD IS TO BE USED
SW2—SET WHEN USING A DECODED OR A SERIAL TYPE OF KEYBOARD

PROGRAMS TO LINK TOGETHER FOR WORKING SYSTEMS:

UNDECODED KEYBOARD—CRTPM.OBJ,CRTPM.SAM,KEYBD.OBJ,PLM51.LIB
DECODED KEYBOARD—CRTPM.OBJ,CRTPM.SAM,DECODE.OBJ,PLM51.LIB
DETACHED KEYBOARD—CRTPM.OBJ,CRTPM.SAM,DETACH.OBJ,PLM51.LIB

* /
$SET (SW1)
$RESET (SW2)
DECLARE LITERALS

DECLARE LLC LITERALLY 'LOCALS.LINESCHANGE';
DECLARE REG LITERALLY 'REGISTER';
DECLARE CURRENTKEY LITERALLY 'CURRENT';
DECLARE SERIAL$SERVICE LITERALLY 'SERBSP';
DECLARE DISPLAY$POINTER LITERALLY 'POINT';
DECLARE SERIAL$INT LITERALLY 'SERINT';
DECLARE TRANSMIT$INT LITERALLY 'TRANINT';
DECLARE CURSERS$COLUMN LITERALLY 'CURSER';
DECLARE LAST$KEY LITERALLY 'LASTKEY';
DECLARE CURSES$COUNT LITERALLY 'COUNT';
DECLARE SCAN$INT LITERALLY 'SCAN';

REGISTER DECLARATIONS FOR THE 8051

DECLARE
    P0 BYTE AT(80H) REG,
    P1 BYTE AT(90H) REG,
    P2 BYTE AT(0A0H) REG,
    P3 BYTE AT(0B0H) REG,
    PSW BYTE AT(0D0H) REG,
    ACC BYTE AT(0E0H) REG,
    B BYTE AT(0F0H) REG,
    SP BYTE AT(81H) REG,
    DPL BYTE AT(82H) REG,
    DPH BYTE AT(83H) REG,
    FCON BYTE AT(87H) REG,
    TCON BYTE AT(88H) REG,
    TMOD BYTE AT(89H) REG,
    TL0 BYTE AT(8AH) REG,
    TL1 BYTE AT(8BH) REG,
    TH0 BYTE AT(8CH) REG,
    TH1 BYTE AT(8DH) REG,
    IE BYTE AT(0A8H) REG,
    IP BYTE AT(0B8H) REG,
    SCON BYTE AT(98H) REG,
    SBUF BYTE AT(99H) REG;
PL/M-51 COMPILER  CRYOCONTROLLER

SELECT

/******* BIT REGISTERS *******/

/******* PSW BITS *******/

DECLARE

14 1

CY BIT AT (007H) REG,
AC BIT AT (006H) REG,
PO BIT AT (005H) REG,
RS1 BIT AT (004H) REG,
RS0 BIT AT (003H) REG,
CV BIT AT (002H) REG,
P BIT AT (001H) REG,

/******* TCON BITS *******/

TF1 BIT AT (8FH) REG,
TF0 BIT AT (8EH) REG,
TR0 BIT AT (8CH) REG,
IE1 BIT AT (8BH) REG,
IT1 BIT AT (8AH) REG,
IE0 BIT AT (89H) REG,
TT0 BIT AT (88H) REG,

/******* IE BITS *******/

EA BIT AT (0AH) REG,
ES BIT AT (O9CH) REG,
ET1 BIT AT (OA8H) REG,
EX1 BIT AT (0A9H) REG,
ETO BIT AT (0AAH) REG,
EX0 BIT AT (0A8H) REG,

/******* IP BITS *******/

PS BIT AT (0BCH) REG,
PT1 BIT AT (0BBH) REG,
FX1 BIT AT (0B8H) REG,
PT0 BIT AT (0B9H) REG,
FX0 BIT AT (0BBH) REG,

/******* P3 BITS *******/

RD BIT AT (0B7H) REG,
WR BIT AT (0B6H) REG,
TI BIT AT (0B5H) REG,
TO BIT AT (0B4H) REG,
INT1 BIT AT (0B3H) REG,
INT0 BIT AT (0B2H) REG,
TXD BIT AT (0B1H) REG,
RXD BIT AT (0B0H) REG,

/******* SCON BITS *******/

SM0 BIT AT (9FH) REG,
SM1 BIT AT (9EH) REG,
SM2 BIT AT (9DH) REG,
REN BIT AT (9CH) REG,
TB8 BIT AT (9BH) REG,
RB8 BIT AT (9AH) REG,
TT BIT AT (99H) REG,
RI BIT AT (98H) REG,
DECLARE UM$SCAN(16) STRUCTURE
(KEY (8) BYTE) CONSTANT

('890-',5CH,5EH,08H,00H,00H,
/* SCAN 0, SHIFT KEY =0; 8,9,0,-,",", BACK SPACE */

'uiop',5BH,'@',0AH,7EH,
/* SCAN 1, SHIFT =0; u,i,o,p,\,\, LINE FEED, DELETE */

'jkl:',00H,00H,'7',
/* SCAN 2, SHIFT =0; j,k,l,;,: RETURN, 7 */

'm',2CH,',',00H,00H,'/',00H,00H,00H,
/* SCAN 3, SHIFT =0; m,COMMA,, */

00H,'azxcvbn',
/* SCAN 4, SHIFT =0; a,z,x,c,v,b,n */

'y',00H,00H,'dgh',
/* SCAN 5, SHIFT =0; y, SPACE, d,f,g,h */

09H,'qwert',00H,
/* SCAN 6, SHIFT =0; q,w,e,r,t */

1BH,'123456',00H,
/* SCAN 7, SHIFT =0; ESC,1,2,3,4,5,6 */

28H,29H,00H,='-',7CH,7EH,08H,00H,
/* SCAN 0, SHIFT =1; (,),=, , , BACK SPACE */

'uiop',00H,00H,00H,7FH,
/* SCAN 1, SHIFT =1; u,i,o,p, LINE FEED, DELETE */

'JKL*',00H,00H,27H,
/* SCAN 2, SHIFT =1; J,K,L,*, RETURN, ' */

'MNC>',00H,3PH,00H,00H,00H,
/* SCAN 3, SHIFT =1; M,<,>,? */

00H,'AZXCVBN',
/* SCAN 4, SHIFT =1; A,Z,X,C,V,B,N */

'y',00H,00H,'DPGH',
/* SCAN 5, SHIFT =1; Y, SPACE, D,F,G,H */

09H,'QWERT',00H,
/* SCAN 6, SHIFT =1; Q,W,E,R,T */

1BH,'!*$%&\',00H),
/* SCAN 7, SHIFT =1; ESC,1",",",$,%& */

$ENDIF
DECLARE

/********************DECLARE VARIABLES********************/

16 1 DECLARE

$IF SW2

INPUT

BIT AT (0B4H) REG,

SENDIF

$IF SW1

CAP$LOCK BIT AT (095H) REG,

SHIFT$KEY BIT AT (096H) REG,

CONTROL$KEY BIT AT (097H) REG,

SENDIF

LOCAL$LINE BIT AT (0B5H) REG,

CLEAR$TO$SEND BIT AT (093H) REG,

DATA$TERMINAL$READY BIT AT (094H) REG;

17 1 DECLARE

$IF SW1

SAME,

VALID$KEY,

KEY0,

LAST$SHIFT$KEY,

LAST$CONTROL$KEY,

LAST$CAP$LOCK,

SENDIF

$IF SW2

REC$WLG,

SYNC,

BY$FIN,

KBD$INT,

ERROR,

SENDIF

NEWS$KEY,

TRANSMIT$TOGGLE,

CURREN$ION,

SERIAL$INT,

SCAN$INT,

TRANSMIT$INT,

ESCSEQ,

VALID$RECEPTION,

LLC,

ENSP) BIT PUBLIC;
PL/M-51 COMPILER

$EJECT

18 1 DECLARE ( I, J, K, ASCII$KEY, TRANSMIT$COUNT, TEMP, SHIFT, CURSER$COL, CURSER$COLUMN, CURSER$ROW, CURSER$COUNT, FIFO, RECEIVE) BYTE PUBLIC;

$IFDEF SW1
19 1 DECLARE LAST$KEY(8) BYTE PUBLIC;
$ENDIF

$IFDEF SW2
19 1 DECLARE LAST$KEY(2) BYTE PUBLIC;
$ENDIF

20 1 DECLARE SERIAL(16) BYTE PUBLIC;

21 1 DECLARE DISPLAY$RAM(7CPH) BYTE AT(1000H) AUXILIARY;

22 1 DECLARE

PARAMETERS$ADDRESS BYTE AT(0000H) AUXILIARY,
COMMANDS$ADDRESS BYTE AT(0001H) AUXILIARY;

23 1 DECLARE ( DISPLAY$RAM$POINTER,
RASTER,
LINE8,
L) WORD PUBLIC;
EXTERNAL PROCEDURE SCANS THE 8 LINES OF THE KEYBOARD AND READS THE RETURN LINES. THE STATUS OF THE 8 RETURN LINES ARE THEN STORED IN INTERNAL MEMORY ARRAY CALLED CURRENTSKEY. THE PROCEDURE CONTROLS 2 STATUS FLAGS; KEY0 AND SAME. KEY0 IS SET IF ALL 8 SCANS READ NO KEY WAS PRESSED. IF ALL 8 SCANS ARE THE SAME AS THE LAST READING OF THE KEYBOARD, THEN SAME IS SET.

PROCEDURE READER: THIS PROCEDURE WRITTEN IS ASSEMBLY LANGUAGE. THE EXTERNAL PROCEDURE FILLS LINE0 SCAN WITH SPACES (2OH ASCII) DURING THE SCROLL ROUTINES.

PROCEDURE BLANK: THIS EXTERNAL PROCEDURE BLANKS FROM THE CURSOR TO THE END OF THE DISPLAY LINE.

PROCEDURE FILL: THIS EXTERNAL PROCEDURE FILLS THE CURSOR LINE WITH SPACES.
PL/M-51 COMPILER CRT CONTROLLER

SELECT

/* PROCEDURE CHECK BAUD RATE: THIS PROCEDURE READS THE THREE PORT PINS ON P1 AND SETS UP THE SERIAL PORT FOR THE SPECIFIED BAUD RATE */

32 1 CHECK$BAUD$RATE:
PROCEDURE;
33 2 $CON=70H; /* MODE 1 */
34 2 TMOD=TMOD OR 20H; /* TIMER 1 AUTO RELOAD */
35 2 TH1=1; /* TIMER 1 ON */
36 2 ES=1; /* ENABLE SERIAL INTERRUPT * /
37 2 ENSP=1; /* SERIAL INTERRUPT MASK FLAG * /
38 3 DO CASE (P1 AND 07H);
39 3 ; /* 00 IS NOT ALLOWED */
40 3 TH1=040H; /* 150 BAUD */
41 3 TH1=0A0H; /* 300 BAUD */
42 3 TH1=0D0H; /* 600 BAUD */
43 3 TH1=0E8H; /* 1200 BAUD */
44 3 TH1=0F4H; /* 2400 BAUD */
45 3 TH1=0FAH; /* 4800 BAUD */
46 3 TH1=0FBH; /* 9600 BAUD */
47 3 END;
48 1 END CHECK$BAUD$RATE;

/* PROCEDURE LOAD CURSER: LOAD CURSER TAKES THE VALUE HELD IN RAM AND LOADS IT INTO THE 8276 CURSER REGISTERS. */

49 1 LOAD$CURSER:
PROCEDURE;
50 2 IF CURLER$CN=1 THEN
51 2 CURSOR$COL=CURSOR$COLUMN;
52 2 EXL=0; /* DISABLE BUFFER INTERRUPT */
53 2 COMMAND$ADDRESS=80H; /* INITIALIZE CURSER COMMAND */
54 2 PARAMETERS$ADDRESS=CURSOR$COL;
55 2 PARAMETERS$ADDRESS=CURSOR$ROW;
56 2 EXL=1; /* ENABLE BUFFER INTERRUPT */
57 1 END LOAD$CURSER;

/* PROCEDURE CARRIAGE$RETURN */

58 1 CARRIAGE$RETURN:
PROCEDURE;
59 2 DISPLAY$RAM$PTER=DISPLAY$RAM$PTER-CURSOR$COLUMN;
60 2 CURSOR$COLUMN=0;
61 2 CURLER%SNC=1;
62 2 CALL LOAD$CURSER;
63 1 END CARRIAGE$RETURN;
PL/M-51 COMPILER  CRTCNTROLLER

SELECT
/* PROCEDURE DOWN CURSOR: THIS PROCEDURE MOVES THE CURSOR DOWN ONE ROW
BY ADDING 1 TO THE CURSOR ROW RAM LOCATION THEN CALL LOAD CURSOR */

64 1 DOWN$CURSOR:
65 2 PROCEDURE;
66 3 IF CURSER$ROW < 18H THEN
67 4 DO;
68 5 CURSERSON=1;
69 6 CURSER$ROW=CURSER$ROW + 1;
70 7 CALL LOAD$CURSER;
71 8 IF DISPLAY$RAM$POINTER < 780H THEN
72 9 DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER+50H;
73 10 ELSE
74 11 DISPLAY$RAM$POINTER=(DISPLAY$RAM$POINTER-780H);
75 12 END;
76 13 CALL FILL;
77 14 DISPLAY$RAM(L)=20H;
78 15 END;
79 16 END DOWN$CURSER;

/* PROCEDURE UP CURSOR: THIS PROCEDURE MOVES THE CURSOR UP ONE ROW
BY SUBTRACTING 1 TO THE CURSOR ROW RAM LOCATION THEN CALL LOAD CURSOR */

81 1 UP$CURSER:
82 2 PROCEDURE;
83 3 IF CURSER$ROW >0 THEN
84 4 DO;
85 5 CURSER$ROW=CURSER$ROW - 1;
86 6 CURSERSON=1;
87 7 CALL LOAD$CURSER;
88 8 IF DISPLAY$RAM$POINTER<50H THEN
89 9 DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER+780H;
90 10 ELSE
91 11 DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER-50H;
92 12 END;
93 13 IF DISPLAY$RAM(L)=0FH THEN /* LOOK FOR END OF LINE*/
94 14 CALL FILL; /* IF TRUE FILL WITH */
95 15 DISPLAY$RAM(L)=20H; /* SPACES */
96 16 END;
97 17 END UP$CURSER;
PL/M-51 COMPILER

CRICONTROLLER

SELECT

/* PROCEDURE RIGHT CURSER: THIS PROCEDURE MOVES THE CURSER RIGHT ONE COLUMN
BY ADDING 1 TO THE CURSER COLUMN RAM LOCATION THEN CALL LOAD CURSER */

98 1 RIGHT$CURSER:
   PROCEDURE;
99 2 IF CURSER$COLUMN < 4FH THEN
100 3 DO;
101 3 CURSER$COLUMN=CURSER$COLUMN + 1;
102 3 CURSERN$ON=1;
103 3 CALL LOAD$CURSER;
104 3 DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER +1;
105 3 END;
106 1 END RIGHT$CURSER;

/* PROCEDURE LEFT CURSER: THIS PROCEDURE MOVES THE CURSER LEFT ONE COLUMN
BY SUBTRACTING 1 TO THE CURSER COLUMN RAM LOCATION THEN CALL LOAD CURSER */

107 1 LEFT$CURSER:
   PROCEDURE;
108 2 IF CURSER$COLUMN >0 THEN
109 3 DO;
110 3 CURSER$COLUMN=CURSER$COLUMN - 1;
111 3 CURSERN$ON=1;
112 3 CALL LOAD$CURSER;
113 3 DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER -1;
114 3 END;
115 1 END LEFT$CURSER;
PL/M-51 Compiler CRT Chamber Controller

SELECT

/* PROCEDURE MOVSCURSER: THIS PROCEDURE IS USED IN CONJUNCTION WITH WORDSTAR IF A ESC F IS RECEIVED FROM THE HOST COMPUTER, THE TERMINAL CONTROLLER WILL READ THE NEXT TWO BYTES TO DETERMINE WHERE TO MOVE THE CURSOR. THE FIRST BYTE IS THE ROW INFORMATION FOLLOWED BY THE COLUMN INFORMATION */

DO WHILE FIFO<4; /* WAIT UNTILL THE MOVSCURSER PARAMETERS*/
END; /* ARE RECEIVED INTO THE FIFO */
TEMP=CURSERSROW;
CURSERSHOW=SERIAL(2);
IF CURSERSHOW>TEMP THEN
   DO:
      L=DISPLAY$RAM$POINTER+((CURSERSHOW-TEMP)*50H);
      IF L>7CH THEN
         DISPLAY$RAM$POINTER=L-700H; /* RAP AROUND TO BEGINNING */
      ELSE
         DISPLAY$RAM$POINTER=L;
      END;
   END;
   IF CURSERSHOW<TEMP THEN
      END;
   ELSE
      END;
   IF CURSERSHOW<TEMP THEN
      END;
   ELSE
      END;
   IF CURSERSHOW>TEMP THEN
      END;
   ELSE
      END;
   TEMP=CURSERSCOLUMN;
   CURSERSCOLUMN=SERIAL(3);
   IF CURSERSCOLUMN>TEMP THEN
      DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER+(CURSERSCOLUMN-TEMP);
   ELSE
      DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER-(TEMP-CURSERSCOLUMN);
   CALL LOADCURSER;
   L=DISPLAY$RAM$POINTER-CURSERSCOLUMN;
   IF DISPLAY$RAM (L)=0FH THEN /* LOOK FOR END PO LINE CHARACTER*/
      DO:
      CALL FILL; /* IF TRUE FILL WITH SPACES */
      DISPLAY$RAM (L)=20H;
      END;
   ELSE
      ES=0;
      DO I=2 TO FIFO-2;
      SERIAL (I)=SERIAL (I+2);
      END;
   ELSE
      END MFVSCURSER;
PLM-51 Compiler CRT Controller

SELECT

/* PROCEDURE ERASE FROM CURSOR TO END OF SCREEN: */

157 1 ERASE FROM CURSOR TO END OF SCREEN:
PROCEDURE;
158 2 CALL BLINE; /* ERASE CURRENT LINE */
159 2 IF CURSER$ROW < 18H THEN
160 3 DO;
161 3 "L=DISPLAY$RAM$POINTER-CURSER$COLUMN$50H; /* GET NEXT LINE */
162 4 DO WHILE (L < 7DH) AND (L <> (LINE0 AND 7FH));
163 4 DISPLAY$RAM(L)=OF1H; /* ERASE UNTIL LINE0 OR */
164 4 L=L+50H; /* END OF DISPLAY RAM */
165 4 END;
166 3 3 L=0;
167 4 DO WHILE L <> (LINE0 AND 7FH); /* ERASE UNTIL LINE0 */
168 4 DISPLAY$RAM(L)=OF1H;
169 4 L=L+50H;
170 4 END;
171 3 END;
172 1 END ERASE FROM CURSOR TO END OF SCREEN;

/* PROCEDURE HOME: THIS PROCEDURE MOVES THE CURSOR TO THE 0,0 POSITION */

173 1 HOME:
PROCEDURE;
174 2 CURSOR$ROW=00;
175 2 CURSER$COLUMN=00;
176 2 CURSOR$COL=1;
177 2 CALL LOAD$CURSER;
178 2 DISPLAY$RAM$POINTER=(LINE0 AND 7FH);
179 1 END HOME;
/* PROCEDURE CLEAR SCREEN */

CLEARSCREEN:
PROCEDURE;
CALL HOME;
CALL ERASE$FROM$CURSOR$TO$END$OF$SCREEN;
END CLEARSCREEN;

/* PROCEDURE SCROLL */

SCROLL:
PROCEDURE;
CALL BLANK;
EXO=0;
/* DISABLE VERTICAL REFRESH INTERRUPT */
IF LINEO=LF80H THEN
LINEO=180H;
ELSE
LINEO=LINEO+50H;
END SCROLL;
/* ENABLE VERTICAL REFRESH INTERRUPT */

/* PROCEDURE LINESPEED */

LINESPEED:
PROCEDURE;
IF CURSORSROW=18H THEN
CALL SCROLL;
ELSE
DO;
CURSORSROW=CURSORSROW+1;
CURSERS$=1;
CALL LOADCURSOR;
END;
END LINESPEED;

IF DISPLAY$RAM$POINTER $77FH THEN
DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER-780H;
ELSE
DISPLAY$RAM$POINTER=DISPLAY$RAM$POINTER+50H;
L=DISPLAY$RAM$POINTER-CURSERSCOLUMN;
IF DISPLAY$RAM$(L)=0FH THEN /* LOOK FOR END OF LINE CHARACTER*/
DO;
CALL FILL; /* IF TRUE FILL WITH SPACES */
DISPLAY$RAM$(L)=20H;
END;
END LINESPEED;
PL/M-51 COMPILER  CRT CONTROLLER

$EJECT

/* Procedure DISPLAY: This procedure will take the byte in RAM labeled RECEIVE and put it into the DISPLAY RAM. */

210 1 DISPLAY:
211 2 Procedure;
212 2 DISPLAY$RAM(DISPLAY$RAM$POINTER)=RECEIVE;
213 2 IF DISPLAY$RAM$POINTER=7CH THEN /* If END of RAM */
214 2 DISPLAY$RAM$POINTER=000H; /* RAP around to beginning */
215 2 ELSE
216 3 DO;
217 3 CURSERS$COLUMN=00H;
218 3 L=DISPLAY$RAM$POINTER;
219 3 IF DISPLAY$RAM(L)=0FH THEN
220 4 DO;
221 4 CALL FILL;
222 4 DISPLAY$RAM(L)=20H;
223 4 END;
224 3 IF CURSERS$ROW=18H THEN
225 3 CALL SCROLL;
226 3 ELSE
227 3 CURSERS$ROW=CURSERS$ROW+1;
228 2 ELSE
229 2 CURSERS$COLUMN=CURSERS$COLUMN+1;
230 2 CURSERS$ROW=1;
231 1 CALL LOADCURSER;
232 1 END DISPLAY;

2-125
PROCEDURE DECIPHER: THIS PROCEDURE DECODES THE HOST COMPUTER'S MESSAGES AND DETERMINES WHETHER IT IS A DISPLAYABLE CHARACTER, CONTROL SEQUENCE, OR AN ESCAPE SEQUENCE THE PROCEDURE THEN ACTS ACCORDINGLY */

DECIPHER:
PROCEDURE;
START$DECIPHER:
VALID$RECEPTION=0;
I=0;
DO WHILE (I<FIFO) AND (SERIAL(I)>1FH) AND (SERIAL(I)<7FH);
RECEIVE$SERIAL(I);
CALL DISPLAY;
I=I+1;
END;
IF I>0 THEN
DO;
ES=0; /* DISABLE SERIAL INTERRUPT WHILE MOVING FIFO */
K=FIFO-I;
DO J=0 TO K; /* MOVE FIFO */
SERIAL(J)=SERIAL(I);
I=I+1;
END;
FIPO=K;
ES=ESP;
VALID$RECEPTION=1;
END;
IF FIPO=0 THEN
DO;
SERIAL$INT=0;
GOTO ENDSDECIPHER;
END;
IF (SERIAL(0)=1BH) THEN
DO;
IF (ESC$SEQ=1) AND (FIPO<2) THEN
GOTO ENDSDECIPHER;
K=(SERIAL(1) AND 5FH)-40H;
IF (K>00H) AND (K<0CH) THEN
DO;
CASE;
END;
ES=0; /* DISABLE SERIAL INTERRUPTS WHILE MOVING FIFO */
PL/M-51 Compiler

CR.CONTROLLER

DO I=0 TO (FIFO-2);
   SERIAL(I)=SERIAL(I+2); /* MOVE FIFO */
END;
FIFO=FIFO-2;
ES=ENS;
VALID$RECEPTION=1;
IF FIFO=0 THEN
DO;
   SERIAL$INT=0;
   GOTO END$DECRYPT;
END;
IF (SERIAL(0) > 07H) AND (SERIAL(0) < 02H) THEN
   DO;
      DO CASE (SERIAL(0) - 06H);
      CALL LEFT$CURSER; /* CTL H */
      CALL LINE$FEED; /* CTL J */
      CALL CLEAR$SCREEN; /* CTL L */
      CALL CARRIAGE$RETURN; /* CTL M */
   END;
      ES=0;
      DO I=0 TO (FIFO-1);
      SERIAL(I)=SERIAL(I+1); /* MOVE FIFO */
   END;
      ES=ENS;
      VALID$RECEPTION=1;
   END;
IF VALID$RECEPTION=0 THEN
   DO;
      ES=0;
      DO I=0 TO (FIFO-1);
      /* IF CHARACTER IS UNRECOGNIZED THEN */
      SERIAL(I)=SERIAL(I+1); /* TRASH IT */
   END;
      ES=ENS;
      FIFO=FIFO-1;
   END;
IF FIFO=0 THEN
   SERIAL$INT=0;
END$DECRYPT;
END $DECRYPT;

2-127
AP-223

PL/M-51 COMPILER CRT CONTROLLER

$EXIT
/* PROCEDURE TRANSMIT- THIS PROCEDURE LOOKS AT THE CLEAR TO SEND PIN FOR AN ACTIVE LOW SIGNAL. ONCE THE MAIN COMPUTER SIGNALS THE 8051 THE ASCII CHARACTER IS PUT INTO THE SERIAL PORT. */

322 1 TRANSMIT:
PROCEDURE;
323 2 IF LOCAL$LINE =1 THEN
324 3 DO;
325 4 DO WHILE (CLEAR$TOSEND=1) OR (TRANSMIT$INT=0);
326 4 END;
327 3 SBUF=ASCII$KEY;
328 3 TRANSMIT$INT=0;
329 3 END;
330 2 ELSE
331 3 DO;
332 3 SERIAL(FIFO)=ASCII$KEY;
333 3 FIFO=FIFO+1;
334 3 SERIAL$INT=1;
335 1 END TRANSMIT;

/* PROCEDURE AUTO$REPEAT: THIS PROCEDURE WILL PERFORM AN AUTO REPEAT FUNCTION AFTER A FIXED DELAY PERIOD */

336 1 AUTO$REPEAT:
PROCEDURE;
337 2 IF NEWSKEY=1 THEN
338 3 DO;
339 3 TRANSMIT$TOGGLE=0;
340 3 TRANSMIT$COUNT=00H;
341 3 CALL TRANSMIT; /* FIRST CHARACTER */
342 3 NEWSKEY=0;
343 3 END;
344 2 ELSE
345 3 DO;
346 4 IF TRANSMIT$COUNT <> 00H THEN
347 4 DO;
348 4 TRANSMIT$COUNT=TRANSMIT$COUNT+1;
349 4 IF TRANSMIT$COUNT=OFFH THEN /* DELAY BETWEEN FIRST CHARACTER AND THE SECOND */
350 5 DO;
351 5 CALL TRANSMIT; /* SECOND CHARACTER */
352 5 END;
353 4 END;
354 3 ELSE
355 4 DO;
356 4 CURSOR$XON=1;
357 4 IF TRANSMIT$TOGGLE = 1 THEN /* 2 VERT FRAMES BETWEEN 3RD TO NTH CHARACTER */
358 4 CALL TRANSMIT; /* 3RD THROUGH NTH CHARACTER */
359 4 TRANSMIT$TOGGLE= NOT TRANSMIT$TOGGLE;
360 4 END;
361 3 END;
362 1 END AUTO$REPEAT;
AP-223

PL/M-51 COMPILER  CRT CONTROLLER

$EJECT

/******************** START MAIN PROGRAM **********************/
/* BEGIN BY PUTTING ASCII CODE FOR BLANK IN THE DISPLAY RAM;*/

363 1 INIT:
DO L=0 TO 7CH;
364 2     DISPLAY$RAM(L)=20H;
365 2 END;
/* INITIALIZE POINTERS, RAM BITS, ETC. */

366 1 ESC$SEQ=0;
367 1 SCAN$INT=0;
368 1 SERIAL$INT=0;
369 1 FIFO=0;
370 1 CURSOR$COUNT=0;
371 1 LLC=0;
372 1 DATA$TERMINAL$READ=1;
373 1 TCON=05H;
374 1 LINED=180H;
375 1 RASTER=180H;
376 1 DISPLAY$TRANSPONTER=0000H;
377 1 TRANSMIT$INT=1;

$IF SW1
378 2 DO I=0 TO 7;
379 2     LAST$KEY(I)=00H;
380 2 END;
381 1 VALID$KEY=0;
382 1 LAST$SHIFT$KEY=1;
383 1 LAST$CONTROL$KEY=1;
384 1 LAST$CAPS$LOCK=1;
$ENDIF

$IF SW2
    KCVFLG=0;
    SYNC=0;
    BFIN=0;
    KBINT=0;
    ERROR=0;
$ENDIF

/******************** INITIALIZE THE 8276 ********************/

385 1 COMMANDADDRESS=00H;  /* SET THE 8276 */
386 1 PARAMETERSADDRESS=4FH;  /* NORMAL ROWS, 80 CHARACTER/ROW */
387 1 PARAMETERSADDRESS=58H;  /* 2 ROW COUNTS PER VERTICAL RETRACE 
25 ROWS PER FRAME */
388 1 PARAMETERSADDRESS=89H;  /* LINE 9 IS THE UNDERLINE POSITION 
10 LINES PER ROW */
389 1 PARAMETERSADDRESS=0F9H;  /* OFFSET LINE COUNTER, NON-TRANSPARENT FIELD ATTRIBUTE */

2-129
PLM-51 COMPILER
CRT CONTROLLER

NON-BLINKING UNDERLINE CURSER, 20 CHARACTER COUNTS PER HORIZONTAL RETRACE */

390 1 TEMP=COMMANDADDRESS;
391 1 CURSERCOLUMN=00H;
392 1 CURBER$ROW=00H;
393 1 CURSERCOL=00H;
394 1 CALL LOADCURSER;
395 1 TEMP=COMMANDADDRESS;
396 1 COMMANDADDRESS=0E0H; /* PRESET 8276 COUNTERS */
397 1 TEMP=COMMANDADDRESS;
398 1 COMMANDADDRESS=23H; /* START DISPLAY */
399 1 COMMANDADDRESS=0AH; /* ENABLE INTERRUPTS */
400 1 TEMP=COMMANDADDRESS;

/* SET UP INTERRUPTS AND PRIORITIES */

401 1 IF SW1
402 1 IP=10H; /* SERIAL PORT HAS HIGHEST PRIORITY */
403 1 IE=2SH; /* ENABLE 8051 EXTERNAL INTERRUPTS */
404 1 $ENDIF

405 1 IF SW2
406 1 IP=10H; /* SERIAL PORT HAS HIGHEST PRIORITY */
407 1 IE=67H; /* ENABLE TIMERO INTERRUPT */
408 1 TM0=05H; /* TIMER 0 EVENT COUNTER */
409 1 TLO=0FH; /* INITIALIZE COUNTER TO FFFF */
410 1 TH0=0FH;
411 1 TR0=1;
412 1 $ENDIF

/* PROCEDURE SCANNER: THIS PROCEDURE SCANS THE KEYBOARD AND DETERMINES IF A SINGLE VALID KEY HAS BEEN PUSHED. IF TRUE THEN THE ASCII EQUIVALENT WILL BE TRANSMITTED TO THE HOST COMPUTER */

413 1 SCANNER:
414 1 EA=1;
415 1 DATA$TERMINAL$READY=0;
416 1 IF CURBER$COUNT=1PH THEN /* PROGRAMMABLE CURSER BLINK */
417 2 D0;
418 2 CURBER$COLUMN=NOT CURBER$COLUMN;
419 2 CURBER$COUNT=00;
420 2 IF CURBER$=00 THEN
421 3 CURBER$COL=7FH;
422 3 CALL LOADCURSER;
423 3 END;
424 3 IF LLC<>LOCAL$LINE THEN /* IF LOCAL/LINE HAS CHANGED STATUS */
425 3 D0;
426 3 IF LOCAL$LINE=0 THEN
427 4 DO;
428 4 ENP=0;
429 4 ES=0;
430 4 END;
431 4 ELSE
432 4 CALL CHECK$BAUD$RATE;
433 4 LLC=LOCAL$LINE;
434 4 END;
435 4 IF SW1
436 4 D0 WHILE SCAN$INT=0; /* WAIT UNTIL VERTICAL RETRACE BEFORE */
437 5 IF SERIAL$INT=1 THEN /* SCANNING THE KEYBOARD */
438 5 CALL DECIPHER;
439 5 END;

2-130
CALL READER;
IF VALID$KEX = 1 AND SAME = 1 AND (LAST$SHIFT$KEY = SHIFT$KEY) AND (LAST$CAP$LOCK = CAP$LOCK) AND (LAST$CONTROLS$KEY = CONTROLS$KEY) THEN
CALL AUTO$REPEAT;
ELSE
DO;
IF KEY = 0 AND SAME = 0 THEN
DO;
TEMP = 0;
DO WHILE LAST$KEY (K) = 0;
K = K + 1;
END;
TEMP = LAST$KEY (K);
DO I = (K + 1) TO 7;
TEMP = TEMP - LAST$KEY (I);
END;
IF TEMP = LAST$KEY (K) THEN
DO;
J = 0;
DO WHILE (TEMP AND 01H) = 0;
TEMP = SHR (TEMP, 1);
J = J + 1;
END;
IF TEMP > 1 THEN
IF CONTROLS$KEY = 0 THEN
ASCII$KEY = (LOW$SCAN (K), KEY (J)) AND 1FH;
ELSE
DO;
ASCII$KEY = (LOW$SCAN (K + 08H), KEY (J));
ELSE
DO;
ASCII$KEY = (LOW$SCAN (K), KEY (J));
IF (CAP$LOCK = 0) AND (ASCII$KEY > 60H) AND (ASCII$KEY < 7BH) THEN
ASCII$KEY = ASCII$KEY - 20H;
IF LIC = 0 THEN
IF ASCII$KEY = 1BH THEN
ESC$SEQ = 1;
ELSE
ESC$SEQ = 0;
END;
END;
ENDIF
LAST$SHIFT$KEY = SHIFT$KEY;
LAST$CAP$LOCK = CAP$LOCK;
LAST$CONTROLS$KEY = CONTROLS$KEY;
VALID$KEY = 1;
NEWS$KEY = 1;
END;
ELSE
DO;
VALID$KEY = 0;
NEWS$KEY = 0;
END;
END;
END;
ELSE
DO;
END;
PL/M-51 COMPILER  CRT CONTROLLER

```plaintext
SEJECT

IFDEF SW2
IF SERIAL$INT=1 THEN
CALL DECRYPT;
IF KBDINT =1 THEN
DO;
IF ERROR =0 THEN
DO;
ASCII$KEY=LIST$KEY(1);
NEW$KEY=1;
CALL AUTO$REPEAT;
KBDINT=0;
END;
ERROR=0;
KBDINT=0;
END;
ENDIF

485 1  GOTO SCANNER;
486 1  END;
```

**MODULE INFORMATION:**

<table>
<thead>
<tr>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size</td>
<td>0860H</td>
</tr>
<tr>
<td>Constant Size</td>
<td>0080H</td>
</tr>
<tr>
<td>Direct Variable Size</td>
<td>2D+00H</td>
</tr>
<tr>
<td>Indirect Variable Size</td>
<td>00H+00H</td>
</tr>
<tr>
<td>Bit Size</td>
<td>10H+00H</td>
</tr>
<tr>
<td>Bit-Addressable Size</td>
<td>00H+00H</td>
</tr>
<tr>
<td>Auxiliary Variable Size</td>
<td>0000H</td>
</tr>
<tr>
<td>Maximum Stack Size</td>
<td>000CH</td>
</tr>
<tr>
<td>Register-Bank (s) Used</td>
<td>0</td>
</tr>
<tr>
<td>1056 Lines Read</td>
<td>0</td>
</tr>
<tr>
<td>0 Program Error(s)</td>
<td>0</td>
</tr>
</tbody>
</table>

END OF PL/M-51 COMPILED

2-132
MC8-51 MACRO ASSEMBLER CKTASM

ISTAS+I MC8-51 MACRO ASSEMBLER V2.1
Object Module Place in $1F1:CHTASM,UBJ
ASSEMBLER INVOKED Bit AS-51 $1FICHASM,SHC

LUC UBJ  LINE  SOURCE

1
2
3
4
5
6
7
PUBLIC BLANK
8
PUBLIC BLINR
9
PUBLIC FILL
10 EXTRN DATA (LINEU,MASTER,POINT,SEAT,SHOOT,CURSEH,COUNT)
11 EXTRN BIT (BYTE,SE08E8,THINT,SCAN)
12
13
----
14 CSEG AT(U3H)  JRESET RASTER TO LINE0 AND SCAN KEYBOARD
15
16
17 J
18
19 J
20
----
21 CSEG AT(U13H)  JNEEDED IF DECODED KEYBOARD IS USED
22
23
----
24 CSEG AT(U23H)  JFILL 8276 RUN BUFFER
25
26
----
27 CSEG
28
29 VENT: PUSH PSA  JPUSS KEY USED BY PLMSI
30
31
32 MOV MASTER,LINEO  JREINITIALIZE MASTER TO LINE0
33
34 MOV MASTER+1,LINEO+1  JCLR 8276 INTERRUPT FLAG
35
36 INC COUNT  JINC CURSOR COUNT REGISTER
37
38 POP VOM  JFOR DEBOUNCE ROUTINE
39
40 POP PSA  JPOP REGISTERS
41
42
43
44 BUFFER: PUSH PSA  JPUSH ALL KEY USED BY PLMSI COVE
45
46
47
48 ACALL UMA  JFILL 8276 RUN BUFFER
49
50
51
52
53 NEII  JPOP REGISTERS
54
55
56
LUC   OBJ   LINE   SOURCE
0056  309904  57   SUBBUF: JNB   U94K,CVER  ;IF TRANSMIT BIT NOT SET THEN CHECK RECEIVE
0057  C299    58   CLK   U94F  ;CLR TRANSMISSION INTERRUPT FLAG
0057  U200    59   SETB   I9AINT  ;SETI TRANS INT FOR PLM51 STATUS CHECK
0058  204818  60   UVERI: JB   VH4,CGWALK  ;IF HI NOT SET GO BACK
0058  L001    61   PUSH   U1
0059  A999    62   MOV   M1,SBUF  ;READ SBUF
0059  L000    63   PUSH   PSK  ;PUSH REGISTERS USED BY PLM51
0059  C000    64   PUSH   ACL
0059  C000    65   PUSH   UOM
0059  C000    66   PUSH   CLK
0059  C200    67   MOV   E8USEG  ;CLR ESC SEQUENCE FLAG
0059  C500    68   MOV   A,#SERIAL  ;GET SERIAL FIFO RAM START LOCATION
0059  C500    69   ADD   A,F1FC  ;AND FIND MOD PAK INTO THE FIFO WE ARE
0059  F8      70   MOV   R0,A  ;PUT IT INTO RI
0059  E9      71   MOV   A,#1  ;CLR BIT 7 OF AC
0059  L227    72   CLK   U37H  ;CLR BIT 7 OF AC
0059  F6      73   MOV   #RI,A  ;PUT DATA IN FIFO
0059  61B2    74   CJNE   A,#1MK,CVERI  ;IF DATA IS NOT A ESC KEY THEN GO OVER
0059  U200    75   SETB   E8USEG  ;SET ESC SEQUENCE FLAG
0059  U200    76   UVERI: INC   F1FC  ;MOV FIFO TO NEXT LOCATION
0059  U200    77   SETB   SEINT  ;SET SERIAL INT BIT FOR PLM51 STATUS CHECK
0059  U000    78   POP   UOM  ;POP REGISTERS
0059  U000    79   POP   ACC
0059  U000    80   POP   PSK
0059  U001    81   POP   O1h
0059  12      82   GOWACKI: RETI
0059  05      83   GBLANK: PUSH   PSK  ;PUSH REG USED BY PLM51
0059  C000    84   PUSH   ACC
0059  C002    85   PUSH   UPL
0059  C003    86   PUSH   UPH
0059  L000    87   PUSH   UOM
0059  65002    88   MOV   UPL,LNEU+1  ;GET LINE0 INFO
0059  65003    89   MOV   UPH,LNE0  ;AND PUT IT INTO DPTh
0059  7501    90   MOV   H0,#50H  ;NUMBER OF CHARACTERS IN A LINE
0059  1401    91   NOTYETI  MOV   A,#20H  ;ASCII SPACE CHARACTER
0059  F0      92   MOVX   @DPTR,A  ;MOV TO DISPLAY RAM
0059  A3      93   INC   UPTR  ;INCH TO NEXT DISPLAY RAM LOCATION
0059  U8FA    94   UJNZ   H0,NOTYETI  ;IF ALL 50H LOCATIONS ARE NOT FILLED
0059  U000    95   POP   U0h  ;GU 00 MORE
0059  D003    96   POP   UPH
0059  U002    97   POP   UPL
0059  U000    98   POP   ACC
0059  U000    99   POP   PSK
0059  U2      100  NET
0059  103+1  101   SEJECT

2-134
LUC OBJ    LINE SOURCE

0VA6 C000 104    ULINE:  PUSH PH    1PUSH REGISTERS USED BY PLM51
0VA6 C0E0 105
0VA6 C002 106    PUSH ACC
0VA6 C003 107    PUSH UPH
0VA6 C000 108    PUSH UOH
00B2 b50003 F 109    MOV UPH,PCINT+1 1GET CURRENT DISPLAY RAM LOCATION
00B5 b50082 F 110    MOV UPL,PCINT+1
00BA 430310 111    URL UPH,#19H 1GET BIT 15 FOR RAM ADDRESS DECODING
00BA A800  F 112    MOV HO,CUR   1GET CURSOR COLUMN IN MAX 10 TELL HOW  
                  1FAR INTO THE RAM YOU ARE
00BB 1420 113   CONT1: MOV A,#20H 1ASCII SPACE CHARACTER
00BF F0  114    MOVX @DPTR,A    1MOV TO DISPLAY RAM
000C A3  115    INC UPH   1INCH TO NEXT DISPLAY RAM LOCATION
000E 68  116    INC NO
000E 830F8 117    CJNE NO,#50H,CONT1 1IF NOT AT THE END OF THE LINE  
                  1CONTINUE
000E 1000 118    POP UOH    1POP REGISTERS
000E 1003 119    POP UPH
000E 1002 120    POP UPL
000E 1000 121    POP PH
000C 12  122    NET
000D C000 123    FILL:  PUSH PH    1PUSH REGISTERS USED BY PLM51
000D C040 124    PUSH ACC
000D C002 125    PUSH UPL
000D C003 126    PUSH UPH
000D C000 127    PUSH UOH
000D 03  128    CLR C
000D 06003 F 129    MOV UPL,#L 1GET BEGINNING OF LINE RAM LOCATION
000D 050082 F 130    MOV UPL,#L+1 1CALCULATED BY PLM51
0011 430310 131    URL UPH,#19H 1GET BIT 15 FOR DISPLAY RAM ADDRESS DECODING
0011 184F  132    MOV HO,#4FH   1SET UP COUNTEN FOR 50M LOCATIONS
0011 A3  133    INC UPH   1GO PAST THE OFH
0011 74C0 134   CONT1: MOV A,#20H 1ASCII SPACE CHARACTER
0011 F0  135    MOVX @DPTR,A    1MOV TO DISPLAY RAM
001E A3  136    INC UPH   1INCH TO NEXT DISPLAY RAM LOCATION
001E 88A  137   DJNZ HO,CUART2 1IF ALL 79 LOCATIONS HAVE NOT BEEN FILLED  
                  1THEN CONTINUE
001E 0000 138    POP UOH    1POP REGISTERS
001E 0003 139    POP UPH
001E 0002 140    POP UPL
001F 0000 141    POP PH
001F 22  142    NET

2-135
<table>
<thead>
<tr>
<th>Line</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>00FA</td>
<td>S50003 F</td>
</tr>
<tr>
<td>00FU</td>
<td>S50002 F</td>
</tr>
<tr>
<td>0100</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0101</td>
<td>A3</td>
</tr>
<tr>
<td>0102</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0103</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0104</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0105</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0106</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0107</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0108</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0109</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>010A</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>010B</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>010C</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>010D</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>010E</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>010F</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0110</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0111</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0112</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0113</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0114</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0115</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0116</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0117</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0118</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0119</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>011A</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>011B</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>011C</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>011D</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>011E</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>011F</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0120</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0121</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0122</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0123</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0124</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0125</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0126</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0127</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0128</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>0129</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>012A</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>012B</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>012C</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>012D</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>012E</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
<tr>
<td>012F</td>
<td>MOVX A, &amp;DPTR</td>
</tr>
</tbody>
</table>

;LOAD XFER POINTER HIGH BYTE
;LOAD XFER POINTER LOW BYTE

;IF INT1 HIGH, THEN UMA IS OVER
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0130 A3</td>
<td>c0b</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0131 A3</td>
<td>c0c</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0132 A3</td>
<td>c10</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0133 A3</td>
<td>c12</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0134 A3</td>
<td>c13</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0135 A0</td>
<td>c14</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0136 A3</td>
<td>c15</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0137 A3</td>
<td>c16</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0138 A0</td>
<td>c17</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0139 A3</td>
<td>c18</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>013A C0</td>
<td>c19</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>013B A3</td>
<td>c20</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>013C A0</td>
<td>c21</td>
<td>THIRTY: MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>013D A3</td>
<td>c22</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>013E A0</td>
<td>c23</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0140 A3</td>
<td>c24</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0141 A0</td>
<td>c25</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0142 A3</td>
<td>c26</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0143 A0</td>
<td>c27</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0144 A3</td>
<td>c28</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0145 A0</td>
<td>c29</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0146 A3</td>
<td>c30</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0147 A0</td>
<td>c31</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0148 A3</td>
<td>c32</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0149 A0</td>
<td>c33</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>014A A3</td>
<td>c34</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>014B A0</td>
<td>c35</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>014C A3</td>
<td>c36</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>014D A0</td>
<td>c37</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>014E A3</td>
<td>c38</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>014F A0</td>
<td>c39</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0150 A3</td>
<td>c40</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0151 A0</td>
<td>c41</td>
<td>FORTY: MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0152 A3</td>
<td>c42</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0153 A0</td>
<td>c43</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0154 A3</td>
<td>c44</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0155 A0</td>
<td>c45</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0156 A3</td>
<td>c46</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0157 A0</td>
<td>c47</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0158 A3</td>
<td>c48</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0159 A0</td>
<td>c49</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>015A A3</td>
<td>c50</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>015B A0</td>
<td>c51</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>015C A3</td>
<td>c52</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>015D A0</td>
<td>c53</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>015E A3</td>
<td>c54</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>015F A0</td>
<td>c55</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0160 A3</td>
<td>c56</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0161 A0</td>
<td>c57</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0162 A3</td>
<td>c58</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0163 A0</td>
<td>c59</td>
<td>MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0164 A3</td>
<td>c60</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>0165 A0</td>
<td>c61</td>
<td>FIFTY: MOVX</td>
<td>A, aOPTN</td>
</tr>
<tr>
<td>0166 A3</td>
<td>c62</td>
<td>INC</td>
<td>UPTR</td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ</td>
<td>LINE</td>
<td>SOURCE</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>014e</td>
<td>A3</td>
<td>310</td>
<td>INC</td>
</tr>
<tr>
<td>014f</td>
<td>00</td>
<td>314</td>
<td>MOVX</td>
</tr>
<tr>
<td>014a</td>
<td>A3</td>
<td>320</td>
<td>INC</td>
</tr>
<tr>
<td>014e</td>
<td>L0</td>
<td>321</td>
<td>MOVX</td>
</tr>
<tr>
<td>014d</td>
<td>A3</td>
<td>322</td>
<td>INC</td>
</tr>
<tr>
<td>014f</td>
<td>ES03</td>
<td>324</td>
<td>LCEK</td>
</tr>
<tr>
<td>014f</td>
<td>p1FC</td>
<td>325</td>
<td>CJNE</td>
</tr>
<tr>
<td>014b</td>
<td>ES02</td>
<td>326</td>
<td>MOV</td>
</tr>
<tr>
<td>014a</td>
<td>ES007</td>
<td>327</td>
<td>CJNE</td>
</tr>
<tr>
<td>014e</td>
<td>750018</td>
<td>328</td>
<td>MOV</td>
</tr>
<tr>
<td>0148</td>
<td>750000</td>
<td>329</td>
<td>MOV</td>
</tr>
<tr>
<td>0132</td>
<td>22</td>
<td>330</td>
<td>RET</td>
</tr>
<tr>
<td>0184</td>
<td>b50300</td>
<td>332</td>
<td>DONE</td>
</tr>
<tr>
<td>0187</td>
<td>b58200</td>
<td>333</td>
<td>MOV</td>
</tr>
<tr>
<td>0188</td>
<td>22</td>
<td>334</td>
<td>RET</td>
</tr>
<tr>
<td>018b</td>
<td>C3</td>
<td>336</td>
<td>UMADNEI</td>
</tr>
<tr>
<td>018c</td>
<td>ES02</td>
<td>337</td>
<td>MOV</td>
</tr>
<tr>
<td>018e</td>
<td>244F</td>
<td>338</td>
<td>ADDU</td>
</tr>
<tr>
<td>01c0</td>
<td>F502</td>
<td>339</td>
<td>MOV</td>
</tr>
<tr>
<td>01c2</td>
<td>50DF</td>
<td>340</td>
<td>JNC</td>
</tr>
<tr>
<td>01c4</td>
<td>v583</td>
<td>341</td>
<td>INC</td>
</tr>
<tr>
<td>01c6</td>
<td>v5v8</td>
<td>342</td>
<td>SJMP</td>
</tr>
<tr>
<td>164</td>
<td>165</td>
<td>ENU</td>
<td></td>
</tr>
</tbody>
</table>
**AP-223**

**MCS-51 MACRO ASSEMBLER CHTASK**

**SYMBOL TABLE LISTING**

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>D</td>
<td>00EH</td>
<td>A</td>
</tr>
<tr>
<td>BLANK</td>
<td>C</td>
<td>00AH</td>
<td>A PUB</td>
</tr>
<tr>
<td>BLINE</td>
<td>C</td>
<td>006H</td>
<td>A PUB</td>
</tr>
<tr>
<td>BUFF</td>
<td>C</td>
<td>003H</td>
<td>A</td>
</tr>
<tr>
<td>CHECK</td>
<td>C</td>
<td>01AH</td>
<td>A</td>
</tr>
<tr>
<td>CUNT1</td>
<td>C</td>
<td>008H</td>
<td>A</td>
</tr>
<tr>
<td>CUNT2</td>
<td>C</td>
<td>00EH</td>
<td>A</td>
</tr>
<tr>
<td>COUNT</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>CURSEP</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>DOUNE</td>
<td>C</td>
<td>00F8H</td>
<td>A</td>
</tr>
<tr>
<td>DMAA</td>
<td>C</td>
<td>00FAH</td>
<td>A</td>
</tr>
<tr>
<td>Damdoe</td>
<td>C</td>
<td>0186H</td>
<td>A</td>
</tr>
<tr>
<td>DONE</td>
<td>C</td>
<td>0184H</td>
<td>A</td>
</tr>
<tr>
<td>DPH</td>
<td>D</td>
<td>0023H</td>
<td>A</td>
</tr>
<tr>
<td>DPL</td>
<td>D</td>
<td>006CH</td>
<td>A</td>
</tr>
<tr>
<td>EIGHTY</td>
<td>C</td>
<td>0141H</td>
<td>A</td>
</tr>
<tr>
<td>ESCSEE</td>
<td>B</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>FIFU</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>FIFTY</td>
<td>C</td>
<td>0165H</td>
<td>A</td>
</tr>
<tr>
<td>FILL</td>
<td>C</td>
<td>00CUH</td>
<td>A PUB</td>
</tr>
<tr>
<td>FURY</td>
<td>C</td>
<td>0151H</td>
<td>A</td>
</tr>
<tr>
<td>GUBACG</td>
<td>C</td>
<td>0084H</td>
<td>A</td>
</tr>
<tr>
<td>L</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>LINEO</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>NUTKE1</td>
<td>C</td>
<td>0097H</td>
<td>A</td>
</tr>
<tr>
<td>OVEN</td>
<td>C</td>
<td>0059H</td>
<td>A</td>
</tr>
<tr>
<td>OVEN1</td>
<td>C</td>
<td>0078H</td>
<td>A</td>
</tr>
<tr>
<td>POINT</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>PSH</td>
<td>D</td>
<td>0000H</td>
<td>A</td>
</tr>
<tr>
<td>RAISEN</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>SBUF</td>
<td>D</td>
<td>0099H</td>
<td>A</td>
</tr>
<tr>
<td>SLAN</td>
<td>B</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>SERBUF</td>
<td>C</td>
<td>0055H</td>
<td>A</td>
</tr>
<tr>
<td>SERIAL</td>
<td>D</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>SERNTR</td>
<td>B</td>
<td>----</td>
<td>EXT</td>
</tr>
<tr>
<td>SELVAT</td>
<td>C</td>
<td>0128H</td>
<td>A</td>
</tr>
<tr>
<td>SIXTY</td>
<td>C</td>
<td>0179H</td>
<td>A</td>
</tr>
<tr>
<td>TEN</td>
<td>C</td>
<td>0113H</td>
<td>A</td>
</tr>
<tr>
<td>THINTY</td>
<td>C</td>
<td>0135H</td>
<td>A</td>
</tr>
<tr>
<td>THNINI</td>
<td>B</td>
<td>----</td>
<td>EAT</td>
</tr>
<tr>
<td>TWENTY</td>
<td>C</td>
<td>0124H</td>
<td>A</td>
</tr>
<tr>
<td>VER1</td>
<td>C</td>
<td>0025H</td>
<td>A</td>
</tr>
</tbody>
</table>

**REGISTER BANK(S) USED: U**

**ASSEMBLY COMPLETE, NO ERRORS FOUND**
ML8-51 KALRU ASSEMBLER KEYBOARD

ISIB-II ML8-51 KALRU ASSEMBLER v2.1
OBJECl MODULE PLACLE IN FP1KEYBD.OBJ
ASSEMBLER INVOKED AT AS851 IF1KEYBU.SHL

LOC OBJ LINE SOURCE
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36 +1 SELECT

;******************************************************************************
;****
;**** SOFTWARE FOR READING AN UNDECICLED KEYBOARD
;****
;****
;******************************************************************************

; THIS CONTAINS THE SOFTWARE NEEDED TO SCAN AN UNDECICLED KEYBOARD
; THIS PROGRAM MUST BE LINKED TO THE MAIN PROGRAMS TO FUNCTION

; MEMORY MAP FOR READING KEYBOARD (USING PUVC)
; ADDRESS FOR KEYBOARD 10FFH TO 17FFH

PUBLIC READEN
EXTERN UATA (LSTKEY)
EXTERN BIT (KEYU,SAME)

;******************************************************************************
;****
;**** "READEN"ROUTINE"
;****
;******************************************************************************

SELECT
AP-223

Keypad

LCS-51 Machine Assembly

LOC WDJ

LINE SOURCE

0000 C000 43 HEADER PUSH #8m ;Push reg used by PLM51
0001 C000 44 PUSH AC
0002 C002 45 PUSH DF
0003 C003 46 PUSH #8m
0004 C004 47 PUSH #2m
0005 C005 48 PUSH #1m
0006 C006 49 PUSH #0m
0007 C007 50 PUSH #3m
0008 C008 51 MOV UPTR,#10FF ;Initialize uptr to keyboard address
0009 C009 52 MOV hi,#60m ;CLR zero counter
0010 C00A 53 MOV k,#8m ;Get keyboard ram pointer
0011 C00B 54 MOV k,#0m ;Initialize loop counter
0012 C00C 55 MOV k,#0m
0013 C00D 56 MOV KI,#000 ;Initialize keyboard status bits
0014 C00E 57 MOV #02m, #00m
0015 C00F 58 MOV #01m, #02m
0016 C010 59 MOV #00m, #03m
0017 C011 5A MOV #02m, #04m
0018 C012 5B MOV #01m, #05m
0019 C013 5C MOV #00m, #06m
001A C014 5D MOV #02m, #07m
001B C015 5E MOV #01m, #08m
001C C016 5F MOV #00m, #09m
001D C017 60 MOV #02m, #0A m
001E C018 61 MOV #01m, #0B m
001F C019 62 MOV #00m, #0C m
0020 C01A 63 MOV #02m, #0D m
0021 C01B 64 MOV #01m, #0E m
0022 C01C 65 MOV #00m, #0F m
0023 C01D 66 SJMP equal ;If equal jump over inch of zero counter
0024 C01E 67 ZEGO INC #1m ;Increment zero counter
0025 C01F 68 CJE #2m, #3m, #4m, #5m
0026 C020 69 CJE #6m, #7m, #8m, #9m
0027 C021 6A CJNE #A, #B, #C, #D
0028 C022 6B CJNE #E, #F, #0, #1
0029 C023 6C CJNE #2, #3, #4, #5
002A C024 6D CJNE #6, #7, #8, #9
002B C025 6E CJNE #A, #B, #C, #D
002C C026 6F CJNE #E, #F, #0, #1
002D C027 70 PDP A ;Pop registers
002E C028 71 PDP B ;Pop registers
002F C029 72 PDP C ;Pop registers
0030 C02A 73 PDP D ;Pop registers
0031 C02B 74 PDP E ;Pop registers
0032 C02C 75 PDP F ;Pop registers
0033 C02D 76 PDP G ;Pop registers
0034 C02E 77 PDP H ;Pop registers
0035 C02F 78 PDP I ;Pop registers
0036 C030 79 PDP J ;Pop registers
0037 C031 7A PDP K ;Pop registers
0038 C032 7B PDP L ;Pop registers
0039 C033 7C PDP M ;Pop registers
003A C034 7D PDP N ;Pop registers
003B C035 7E PDP O ;Pop registers
003C C036 7F PDP P ;Pop registers
003D C037 80 PDP Q ;Pop registers
003E C038 81 PDP R ;Pop registers
003F C039 82 PDP S ;Pop registers
0040 C03A 83 PDP T ;Pop registers
0041 C03B 84 PDP U ;Pop registers
0042 C03C 85 PDP V ;Pop registers
0043 C03D 86 PDP W ;Pop registers
0044 C03E 87 PDP X ;Pop registers
0045 C03F 88 PDP Y ;Pop registers
0046 C040 89 PDP Z ;Pop registers
0047 C041 8A PDP #8m ;Pop registers
0048 C042 8B PDP #7m ;Pop registers
0049 C043 8C PDP #6m ;Pop registers
004A C044 8D PDP #5m ;Pop registers
004B C045 8E PDP #4m ;Pop registers
004C C046 8F PDP #3m ;Pop registers
004D C047 90 PDP #2m ;Pop registers
004E C048 91 PDP #1m ;Pop registers
004F C049 92 PDP #0m ;Pop registers
0050 C04A 93 PDP #3m ;Pop registers
0051 C04B 94 PDP #2m ;Pop registers
0052 C04C 95 PDP #1m ;Pop registers
0053 C04D 96 PDP #0m ;Pop registers
0054 C04E 97 PDP #3m ;Pop registers
0055 C04F 98 PDP #2m ;Pop registers
0056 C050 99 PDP #1m ;Pop registers
0057 C051 9A PDP #0m ;Pop registers
0052 C052 9B PDP #3m ;Pop registers
0053 C053 9C PDP #2m ;Pop registers
0054 C054 9D PDP #1m ;Pop registers
0055 C055 9E PDP #0m ;Pop registers
0056 C056 9F PDP #3m ;Pop registers
0057 C057 A0 PDP #2m ;Pop registers
0058 C058 A1 PDP #1m ;Pop registers
0059 C059 A2 PDP #0m ;Pop registers
005A C05A A3 PDP #3m ;Pop registers
005B C05B A4 PDP #2m ;Pop registers
005C C05C A5 PDP #1m ;Pop registers
005D C05D A6 PDP #0m ;Pop registers
005E C05E A7 PDP #3m ;Pop registers
005F C05F A8 PDP #2m ;Pop registers
0060 C060 A9 PDP #1m ;Pop registers
0061 C061 AA PDP #0m ;Pop registers
AP-223

MCS-51 HALRU ASSEMBLER KEYBD

SYMBOL TABLE LISTING

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALC</td>
<td>AUCH</td>
<td>0VEUH</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>BAC</td>
<td>AUCH</td>
<td>0V3AH</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>D'M</td>
<td>AUCH</td>
<td>0V85H</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>DPL</td>
<td>AUCH</td>
<td>0V82H</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>EQual</td>
<td>AUCH</td>
<td>0V2EH</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>KEYT</td>
<td>AUCH</td>
<td>-----</td>
<td>EXT</td>
</tr>
<tr>
<td>LUTAKT</td>
<td>AUCH</td>
<td>-----</td>
<td>EXT</td>
</tr>
<tr>
<td>MURE</td>
<td>AUCH</td>
<td>0V1uH</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>NISAME</td>
<td>AUCH</td>
<td>0V4uH</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>PSN</td>
<td>AUCH</td>
<td>0V0uH</td>
<td>PUB</td>
</tr>
<tr>
<td>READER</td>
<td>AUCH</td>
<td>0V50H</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
<tr>
<td>SAMT</td>
<td>AUCH</td>
<td>-----</td>
<td>EXT</td>
</tr>
<tr>
<td>UNDECoded_KEYBOARD</td>
<td>SEG</td>
<td>0V5uH</td>
<td>KEL=LINII</td>
</tr>
<tr>
<td>ZERU</td>
<td>AUCH</td>
<td>0V29H</td>
<td>SEG=UNDECoded_KEYBOARD</td>
</tr>
</tbody>
</table>

REGISTER BANK(S) USED: U

ASSEMBLY COMPLETE, NO ERRORS FOUND
MCS-51 HALRU ASSEMBLER  DECIDE

ISIS-11 MCS-51 HALRU ASSEMBLER V2.1
OBJECT MODULE PLACED IN IFI:DECODE.UBJ
ASSEMBLER INVOKED BY: ASPS1 OF IIE:ECODE.SKC

LUC OBJ LIST SOURCE
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

;******************************************************************************************************************
;******************************************************************************************************************
;******************************************************************************************************************
;******************************************************************************************************************

PUBLIC DETACH
EXTERN DATA (L81KEY)
EXTERN dIT (KBINT)

;******************************************************************************************************************
;******************************************************************************************************************
;******************************************************************************************************************
;******************************************************************************************************************

;"DECODE" INTERRUPT ROUTINE FOR DECODED KEYBOARDS

+1 SELECT
AP-223

PLS-51 MACRO ASSEMBLER  DECODE

LUC JJBJ LIST SOURCE

21
22  DELOUEU_KEYBAND SEGMENT CODE
24  SEG DECODE_KEYBAND
30
0000 C000 31  DELALSH: PUSH PSN  ;PUSH REGISTERS
0002 C002 32  PUSH LPL  ;USED BY PLMSI
0004 003 33  PUSH LFM
0006 00E0 34  PUSH ACL
0008 900FF 35  MCV DF1A,1800F6  ;ADDRESS FOR KEYBOARD
000A 40 36  CLK A
000C 43 37  MOV C A,AA+PTRN  ;FETCH ASCII BYTE
000E F500 38  MOV LSKEY+I,A  ;MOV TO MEMORY TO BE READ BY PLMSI
000F 0200 39  SET8 NBLINT  ;LET PLMSI KNOW THERE IS A BYTE
0011 75CF 40  MOV IFO,100FF  ;SET COUNTER TO FFFFH SO INTERRUPT
0014 750F 41  MOV ILV,100FF  ;ON THE NEXT FALLING EDGE UP TO
0017 00C0 42  POP ACC
0019 003 43  POP DFH  ;PUP REGISTERS
001B 002 44  POP UPL
001D 000 45  POP PSN
001F 32 46  CLI
47
48
49
50
51 END
AP-223

MCS-51 HALRU ASSEMBLER  DECODE

SYMBOL TABLE LISTING

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALC</td>
<td>D ALC</td>
<td>0VEUH</td>
<td>A</td>
</tr>
<tr>
<td>DECODED_KEYBOARD</td>
<td>C SEG</td>
<td>0V2UH</td>
<td>REL=ALU</td>
</tr>
<tr>
<td>DETACH</td>
<td>C ALC</td>
<td>0VEUH</td>
<td>REL=DECoded_KEYBOARD</td>
</tr>
<tr>
<td>DPH</td>
<td>D ALC</td>
<td>0V8UH</td>
<td>A</td>
</tr>
<tr>
<td>DPL</td>
<td>D ALC</td>
<td>0V82H</td>
<td>A</td>
</tr>
<tr>
<td>KeQIN1</td>
<td>P ALC</td>
<td>------</td>
<td>EX</td>
</tr>
<tr>
<td>LSTKEY</td>
<td>D ALC</td>
<td>------</td>
<td>EX</td>
</tr>
<tr>
<td>PSK</td>
<td>D ALC</td>
<td>0UDUH</td>
<td>A</td>
</tr>
<tr>
<td>TH0</td>
<td>D ALC</td>
<td>0UBUH</td>
<td>A</td>
</tr>
<tr>
<td>TLO</td>
<td>D ALC</td>
<td>0UBAH</td>
<td>A</td>
</tr>
</tbody>
</table>

REGISTER BANK(S) USED 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
**AP-223**

```
MLS-51 KALRU ASSEMBLE DETACH

ISIS-II MLS-51 PACHL ASSEMBLER V2.1
Object Module Placed In FPIIDETACH.URJ
Assembler Invoked With AS-S1 FPIIDETACH.SNC

LUC UBJ LINE SOURCE

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

15 // THIS CONTAINS THE SOFTWARE NEEDED TO PERFORM A SOFTWARE SERIAL
16 // PORT FOR SERIAL KEYBOARDS AND DETACHABLE KEYBOARD. THIS PROGRAM MUST
17 // BE LINKED TO THE MAIN PROGRAMS FOR USE.
18
19 $select

2-147
```
MC68010 HCLU ASSEMBLER  DETACH

LOC  WHJ  LINE  SOURCE

20  ;
21  ;
22  ;
23  INPUT EQU TV
24  ;
25  ;
26  ;
27  ;
28  ;
29  ;
30  ;
31  ;
32  ;
33  ;
34  ;
35  ;
36  ;
37  ;
38  ;
39  ;
40  ;
41  Baud  Start Bit Detect  Message Detect
42  110  OEF40H  ODF45H
43  ;
44  ;
45  ;
46  ;
47  ;
48  ;
49  ;
50  ;
51  +1 SELECT

Baud  used with detachable keyboards

Timex u load values for different baud rates

Message detect
**MC6801 HALU ASSEMBLER**

**DETECT**

---

**DETECTABLE KEYBOARD SEGMENT CODE**

`SEG DETACHABLE_KEYBOARD`

---

0000 0000h 60  `DETECT: PUSH F3h`
0000 0000h 61  `PUSH ACL`
0000 0004h 67  `JB MCvFLG,VALID`
0000 0007h 60  `JB IMPL,M8I`
0000 000Ah 64  `SEIB MCvFLG`
0000 000Ch 70  `MOV I'0,'SIANT1`
0000 000Fh 71  `MOV TLU,'SIANTU`
0000 0012h 72  `MOV A,IPD`
0000 0014h 73  `CLK VELC`
0000 0016h 74  `MOV IM4,C,A`
0000 0018h 75  `SJMP FINI`
0000 001Ah 76  `;
0000 001Ch 77  `VALID: JB SYNC,AATGII`
0000 0020h 76  `JB IMPL,M8I`
0000 0023h 79  `SEIB SYNC`
0000 0026h 80  `MOV LS1KEY,#80M`
0000 0029h 81  `MOV TH0,#MESSAGE1`
0000 002Ch 82  `MOV TLU,#MESSAGE0`
0000 002Fh 83  `SJMP FINI`
0000 0032h 84  `;
0000 0035h 85  `NXBITI MOV I'0,#MESSAGE1`
0000 0038h 86  `MOV TLU,#MESSAGE0`
0000 003Bh 87  `JB BYFIN,STUP`
0000 003Eh 88  `MOV A,LS1KEY`
0000 0041h 89  `MOV C,IMPL`
0000 0044h 90  `HRC A`
0000 0047h 91  `MOV LS1KEY,A`
0000 004Ah 92  `JNC FINI`
0000 004Dh 93  `SEIB BYFIN`
0000 0050h 94  `CLK VEF7`
0000 0053h 95  `MOV LS1KEY+1,A`
0000 0056h 96  `FINI: POP ACC`
0000 0059h 97  `POP PSA`
0000 005Ch 98  `RET7`
MCS-51 MACRO ASSEMBLER DETACH

LUC ORJ LINE SOURCE

064A 30A4V5 100 STOP: JNO AHPL,ERH ; IF NOT 1 THEN NOT A VALID STOP BIT
044U 0200 F 101 SE1B NBUINT ; TELL PLM A BYTE IS READY
044P 020000 F 102 JMP NS1 ; AND GO BACK TO MAIN PROGRAM

0552 0200 F 103 ;
0554 0200 F 104 ERH1 SE1B ERHCK ; CLEAR FLAGS
0556 0200 F 105 NS1: CLR NCVFHL ;
0558 0200 F 106 CMR SNC
055A 0200 F 107 CMR BRYIN
055C 0202 F 108 MOV A,A
055E 0209 F 109 SE1B VED2 ; SET TIMER 0 TO COUNTING MODE

0661 75dCFF 110 MOV IMLO,A
0663 75dAFF 111 MOV IMHi,AOFFH ; SET COUNTER TO FFFF SO INTERRUPT
0666 b000 112 SJMP FIN

1250 END
**AP-223**

**MLS-51 MCMU ASSEMBLER**

**SYMBOL TABLE LISTING**

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>ACUR</td>
<td>0026H</td>
<td>A</td>
</tr>
<tr>
<td>BYFIA</td>
<td>ACUR</td>
<td>-----</td>
<td>EXI</td>
</tr>
<tr>
<td>DETACH</td>
<td>ACUR</td>
<td>0020H</td>
<td>R REL=UNIT SE=DETACHABLE_KEYBOARD</td>
</tr>
<tr>
<td>DETACHABLE_KEYBOARD</td>
<td>SEG</td>
<td>0068H</td>
<td>R REL=UNIT SE=DETACHABLE_KEYBOARD</td>
</tr>
<tr>
<td>ENR</td>
<td>ACUR</td>
<td>0032H</td>
<td>R REL=UNIT SE=DETACHABLE_KEYBOARD</td>
</tr>
<tr>
<td>ERUR</td>
<td>ACUR</td>
<td>-----</td>
<td>EXI</td>
</tr>
<tr>
<td>FIN1</td>
<td>ACUR</td>
<td>0045H</td>
<td>R REL=UNIT SE=DETACHABLE_KEYBOARD</td>
</tr>
<tr>
<td>INPUT</td>
<td>ACUR</td>
<td>0066H.4A</td>
<td></td>
</tr>
<tr>
<td>KBINIT</td>
<td>ACUR</td>
<td>-----</td>
<td>EXI</td>
</tr>
<tr>
<td>LSTDINIT</td>
<td>ACUR</td>
<td>-----</td>
<td>EXI</td>
</tr>
<tr>
<td>MESSAGES</td>
<td>LMB</td>
<td>0000H</td>
<td>A</td>
</tr>
<tr>
<td>MESSAGE1</td>
<td>LMB</td>
<td>0050H</td>
<td>A</td>
</tr>
<tr>
<td>MDTUNIT</td>
<td>ACUR</td>
<td>002DH</td>
<td>R REL=UNIT SE=DETACHABLE_KEYBOARD</td>
</tr>
<tr>
<td>PWD</td>
<td>ACUR</td>
<td>0000H</td>
<td>A</td>
</tr>
<tr>
<td>RCVFLG</td>
<td>ACUR</td>
<td>-----</td>
<td>EXI</td>
</tr>
<tr>
<td>RST</td>
<td>ACUR</td>
<td>0054H</td>
<td>A</td>
</tr>
<tr>
<td>START1</td>
<td>LMB</td>
<td>0000H</td>
<td>A</td>
</tr>
<tr>
<td>STAMT1</td>
<td>LMB</td>
<td>00F4H</td>
<td>A</td>
</tr>
<tr>
<td>STOP</td>
<td>ACUR</td>
<td>004AH</td>
<td>R REL=UNIT SE=DETACHABLE_KEYBOARD</td>
</tr>
<tr>
<td>SYNC</td>
<td>ACUR</td>
<td>-----</td>
<td>EXI</td>
</tr>
<tr>
<td>TU</td>
<td>ADDR</td>
<td>0000H.4A</td>
<td></td>
</tr>
<tr>
<td>TH0</td>
<td>ACUR</td>
<td>0080H</td>
<td>A</td>
</tr>
<tr>
<td>TLO</td>
<td>ACUR</td>
<td>008AH</td>
<td>A</td>
</tr>
<tr>
<td>TM0</td>
<td>ACUR</td>
<td>0080H</td>
<td>A</td>
</tr>
<tr>
<td>VALID</td>
<td>ADDR</td>
<td>001AH</td>
<td>R REL=UNIT SE=DETACHABLE_KEYBOARD</td>
</tr>
</tbody>
</table>

**REGISTER BANK(S) USED: 0**

**ASSEMBLY COMPLETE, NO ERRORS FOUND**
APPENDIX B
REFERENCES

Interfacing the 82786 Graphics Coprocessor to the 8051

RICK SCHUE
REGIONAL APPLICATIONS SPECIALIST
INDIANAPOLIS, INDIANA
Interfacing the 82786 to the 8051 presents some interesting challenges, but can be accomplished with a little additional logic and software. Since the 82786 looks like a DRAM controller to the host CPU, wait states are often required when accessing the coprocessor. Since wait states are not supported by the 8051, latching transceivers and dummy read and write cycles are used to communicate with the 82786. Byte swapping is also required in the external logic to allow the 8 bit 8051 to read and write the 16 bit graphics memory supported by the 82786. This byte swapping is accomplished with the latching transceivers as well. All of the control logic is implemented in an Intel 5C060 EPLD, allowing the entire interface to fit into three 24 pin DIPs.

HARDWARE

Figure 1 shows the interface between the 8051 bus and the 82786. Figure 2 shows a typical 8051 CPU design needed to complete the circuit. In this design the 82786 is mapped into an 8K byte window in 8051 data memory space. The upper address bits are used as a “page select” and are provided by I/O pins on the 8051. The 5C060 EPLD contains the control logic for the transceivers and address decoding for the 82786. An equivalent circuit for the EPLD is shown in Figure 3; the “ADF” file is shown in Figure 4. The 82786 data memory is mapped into one 8K block (A000H–BFFEH), the 82786 registers are mapped into another (8000H–807EH), and the transceivers are mapped into a third block of memory (C000H–C001H).

OPERATION

Operation of the interface is as follows. For reading the graphics memory, the 8051 sets the upper address bits (PORT 1.0–1.3) and then performs a dummy read operation to the desired location in graphics memory (A000H thru BFFEH). The dummy read cycle provides the address and RD/WR information to the 82786, which runs a cycle and deposits the 16 bit result into the latching transceivers at the end of the read cycle, as indicated by SEN. This event clears the BUSY flip flop in the EPLD. When the BUSY signal goes inactive, the 8051 reads the low byte from the latching transceiver at address C000H and the high byte free address C001H.

For write cycles, the 8051 writes the low byte of the word into the latch at address C000H and the high byte into address C001H. Next the upper address bits are set with PORT1 and a dummy write cycle is performed in graphics memory at the desired address (A000H–BFFEH). Like in the read example, the 82786 runs a memory cycle at this point, enabling the outputs of the latching transceivers at the proper time in the write cycle, as indicated by SEN going active.

Accessing the registers inside the 82786 is done in exactly the same fashion, except that the 82786 is addressed in locations 8000H through 807EH. This causes the EPLD to drive the M/IO pin low during these cycles.

DESIGN NOTES

74F543’s are used for the latching transceivers in this design, although 74HCT646’s could be used to reduce the total power consumption. Some changes to the EPLD would be required in this case. The interface assumes that all memory accesses to the 82786 are word references; accordingly, BHE is grounded at the 82786. All addresses generated by the 8051 must be even byte addresses, the only byte operations allowed are the reads and writes to the latching transceivers. The design shown here incorporates hardware workarounds for the earlier “C-step” 82786; the current “D-step” part will work in the design as well. Additional information regarding the 82786 can be found in the “82786 Graphics Coprocessor User’s Manual”, Intel publication number 231933.
Figure 3. 8051/82786 Control EPLD (5C060-55) Equivalent Circuit
INTEL
August 11, 1987
1-003
0
5C060
8051/82786 Control Logic for 8051 Demo Board
786 I/O 8000H-807FH
786 Memory A000H-BFFFH
Registers C000H,C001H
OPTIONS: TURBO=ON
PART: 5C060
INPUTS: A15@23,A14@14,A13@11,A0@2, RD/@10, WR/@9, SEN@8, CKK
OUTPUTS: 786CS/@15, 786RD/16, OEH/@17, LEH/@18, OEL/@19, LEL/@20, LEW/@21, OEW/@22, READ@4, BUSY@3

NETWORK:
a15 = INP (A15)
a14 = INP (A14)
a13 = INP (A13)
a0 = INP (A0)
rdn = INP (RD/)
wrn = INP (WR/)

786csn = NAND (a15,a14n)
786CS/ = CONF (786csn,VCC)
786rdn = OR (786csn,rdn)
786RD/ = CONF (786rdn,VCC)
hibankn = NAND (a15,a14,a13n,a0)
lobankn = NAND (a15,a14,a13n,a0n)
oehn = OR (hibankn,rdn)
OEH/ = CONF (oehn,VCC)
lehn = OR (hibankn,wrn)
LEH/ = CONF (lehn,VCC)
oeln = OR (lobankn,rdn)
OEL/ = CONF (oeln,VCC)
leln = OR (lobankn,wrn)
LEL/ = CONF (leln,VCC)
oewn = NAND (sen,write)
OEW/ = CONF (oewn,VCC)
lew = AND (sen,read)
q0 = NORF (lew,clk,GND,GND)
q1 = NORF (q0,clk,GND,GND)
q2 = NORF (q1,clk,GND,GND)
q2n = NOT (q2)
lewn = NAND (q1,q2n)
LEW/ = CONF (lewn,VCC)
786wr = NOR (786csn,wrn)
786rd = NOT (786rdn)
READ,read = SOSF (786rd,clk,786wr,GND,GND,VCC)
write = NOT (read)
786rdwr = OR (786rd,786wr)
BUSY = SONF (786rdwr,clk,SEN,GND,GND,VCC)
END$
Interfacing the Densitron LCD to the 8051

RICK SCHUE
REGIONAL APPLICATIONS SPECIALIST
INDIANAPOLIS, INDIANA
INTRODUCTION

This application note details the interface between an 80C31 and a Densitron two row by 24 character LM23A2C24CBW display. This combination provides a very flexible display foot format (2x24) and a cost effective, low power consumption microcontroller suitable for many industrial control and monitoring functions.

Although this applications brief concentrates on the 80C31, the same software and hardware techniques are equally valid on other members of the 8051 family, including the 8031, 8751, and the 8044.

HARDWARE DESIGN

The LCD is mapped into external data memory, and looks to the 80C31 just like ordinary RAM. The register select (RS) and the read/write (R/W) pins are connected to the low order address lines A0 and A1. Connecting the R/W pin to an address line is a little unorthodox, but since the R/W line has the same set-up time requirements as the RS line, treating the R/W pin as an address kept this pin from causing any timing problems.

The enable (E) pin of the LCD is used to select the device, and is driven by the logical OR of the 80C31's RD and WR signals AND'ed with the MSB of the address bus. This maps the LCD into the upper half of the 64 KB external data space. If this seems a little wasteful, feel free to use a more elaborate address decoding scheme.

With the address decoding shown in the example, the LCD is mapped as follows:

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>Read/Write?</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000H</td>
<td>Write Command to LCD</td>
<td>Write Only</td>
</tr>
<tr>
<td>8001H</td>
<td>Write Data to LCD</td>
<td>Write Only</td>
</tr>
<tr>
<td>8002H</td>
<td>Read Status from LCD</td>
<td>Read Only</td>
</tr>
<tr>
<td>8003H</td>
<td>Read Data from LCD</td>
<td>Read Only</td>
</tr>
<tr>
<td>8004H</td>
<td>No Access</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFFFH</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Undefined results may occur if the software attempts to read address 8000H or 8001H, or write to address 8002H or 8003H.

TIMING REQUIREMENTS

The timing requirements of the Densitron LCD are a little slow for a full speed 80C31. The critical timing parameters are the enable pulse width (PW E) of 450 ns, and the data delay time during read cycles (tDDR) of 320 ns. The 80C31 is available at clock speeds up to 16 MHz, but at this speed these parameters are violated. Since the 80C31 lacks a READY pin, the only way to satisfy the LCD timing requirements is to slow the clock down to 10 MHz or lower. A convenient crystal frequency is 7.3728 MHz since it allows all standard baud rates to be generated with the internal timers.

SOFTWARE

The code consists of a main module and a set of utility procedures that talk directly to the LCD. This way the application code does not have to be concerned with where the LCD is mapped, or the exact bit patterns needed to control it. The mainline consists of a call to initialize the LCD, and then it writes a message to the screen, waits, and then erases it. It repeats this indefinitely.

The utility procedures include functions to initialize the display, send data and ads to the LCD, home the cursor, clear the display, set the cursor to a given row and column, turn the cursor on and off, and print a string of characters to the display. Not all the functions are used in the software example.

REFERENCES

INTEL Embedded Controller Handbook, 210918
INTEL PL/M-51 User’s Guide, 121966
DENSITRON Catalog LCDMD-C
Main_module: DO;

Delay: PROCEDURE (count) EXTERNAL;
  DECLARE count WORD;
END Delay;

Initialize_LCD: PROCEDURE EXTERNAL;
END Initialize_LCD;

Clear display: PROCEDURE EXTERNAL;
END Clear_display;

LCD_print: PROCEDURE EXTERNAL;
END LCD_print;

DECLARE LCD_buffer \(48\) BYTE PUBLIC,
  sign_on_message \(\ast\) BYTE CONSTANT
  ('INTEL 8051 DRIVES LCD -'
   '2 ROWS BY 24 CHARACTERS'),
  i BYTE;

/* This is the start of the program */

/* Initialize the LCD */
CALL Initialize_LCD;
CALL Clear_display;

/* Now enter an endless loop to display the message */
DO WHILE 1;

/* Copy the message to the buffer */
DO i = 0 to 47;
  LCD_buffer(i) = sign_on_message(i);
END;

/* Now print out the buffer to the LCD */
CALL LCD_print;

/* wait a while */
CALL Delay(2000);

/* now clear the screen */
CALL Clear_display;

END;  /* of DO WHILE */

END Main_module;

Main Module
LCD_IO_MODULE: DO;

DECLARE LCD_buffer (48) BYTE EXTERNAL,
    LCD_command BYTE AT (08000H) AUXILIARY,
    LCD_data BYTE AT (08001H) AUXILIARY,
    LCD_status BYTE AT (08002H) AUXILIARY,
   LCD_busy LITERALLY '1000$0000B',
    i BYTE;

Delay: PROCEDURE (msec) PUBLIC;

    /* This procedure causes a delay of n msec */

DECLARE msec WORD,
    i WORD;

IF msec > 0 THEN DO;
    DO i = 0 to msec - 1;
        CALL Time(5); /* .2 msec delay */
    END;
END Delay;

LCD_out: PROCEDURE (char) PUBLIC;

DECLARE char BYTE;

    /* wait for LCD to indicate NOT busy */
    DO WHILE (LCD_status AND LCD_busy) <> 0;
    END;

    /* now send the data to the LCD */
    LCD_data = char;

END LCD_out;

LCD Driver Module
LCD_command_out: PROCEDURE (char) PUBLIC;

DECLARE char BYTE;

/* wait for LCD to indicate NOT busy */
DO WHILE (LCD_status AND LCD_busy) <> 0;
END;

/* now send the command to the LCD */
LCD_command = char;
END LCD_command_out;

Home_cursor: PROCEDURE PUBLIC;

CALL LCD_command_out(0000$010B);
END Home_cursor;

Clear_display: PROCEDURE PUBLIC;

CALL LCD_command_out (0000$001B);
END Clear_display;

Set_cursor: PROCEDURE (position) PUBLIC;

DECLARE position BYTE;

IF position > 47 THEN position = 47;
IF position < 24 THEN CALL LCD_command_out(080H + position);
ELSE CALL LCD_command_out(0COH + (position - 24));
END Set_cursor;

Cursor_on: PROCEDURE PUBLIC;

CALL LCD_command_out(0000$111B);
END Cursor_on;

Cursor_off: PROCEDURE PUBLIC;

CALL LCD_command_out(0000$1100B);
END Cursor_off;
LCD_print: PROCEDURE PUBLIC;

/* This procedure copies the contents of the LCD_buffer to the display */

CALL Set_cursor(0);
DO i = 0 to 23;
   CALL LCD_out(LCD_buffer(i));
END;
CALL Set_cursor(24);
DO i = 24 to 47;
   CALL LCD_out(LCD_buffer(i));
END;

END LCD_print;

Initialize_LCD: PROCEDURE PUBLIC;

CALL Delay(100);
CALL LCD_command_out(38H); /* Function Set */
CALL LCD_command_out(38H);
CALL LCD_command_out(06H); /* entry mode set */
CALL Clear_display;
CALL Home_cursor;
CALL Cursor_off;
CALL Set_cursor(0);

END Initialize_LCD;

END LCD_IO_Module;

LCD Driver Module (Continued)
32-Bit Math Routines for the 8051

RICK SCHUE
REGIONAL APPLICATIONS SPECIALIST
INDIANAPOLIS, INDIANA
Here are some easy to use 16- and 32-bit math routines that take the pain out of calculations such as PID loops, A/D calibration, linearization calculations and anything else that requires 32-bit accuracy.

The package is written to interface with PL/M-51. Parameters are passed as 16-bit words to the routines, which perform operations on a 32-bit “accumulator” resident in memory. The following functions are performed:

**Load_16 (word_param)**
Loads a 16-bit RD into the low half of the 32-bit “accumulator”, zeros upper 16 bits of accumulator.

**Load_32 (word_hi,word_lo)**
Loads word_hi into upper 16 bits of accumulator, word_lo into lower 16 bits.

**Low_16**
Returns the lower 16 bits of the accumulator, bits 0 through 15.

**Mid_16**
Returns the middle 16 bits of the accumulator, bits 8 through 23.

**High_16**
Returns the upper 16 bits of the accumulator, bits 16 through 31.

**Mul_16 (word_param)**
Multiplies the 32-bit accumulator by the 16-bit word supplied, result left in accumulator.

**Div_16 (word_param)**
Divides the 32-bit accumulator by the 16-bit word supplied, result left in accumulator.

**Add_16 (word_param)**
Adds the 16-bit word supplied to the 32-bit accumulator.

**Sub_16 (word_param)**
Similar to Add_16 but for subtraction.

**Add_32 (word_hi,word_lo)**
Forms a 32-bit value for word_hi and word_lo and adds it to the accumulator.

**Sub_32 (word_hi,word_lo)**
Similar to Add_32 but for subtraction.

**APPLICATION**

Typical applications have 16-bit “input” values and produce 16-bit “output” values, but require 32-bit values for intermediate results. An example would be reading a 12-bit A/D, performing some gain and offset calculation on the raw A/D data to produce a calibrated 16-bit result. Doing this is a simple task with this math package.

```plaintext
CALL Load_16 (AD_value);
CALL Add_16 (offset_value);
CALL Mul_16 (gain_factor);
/* gain is in units of 1/256 */
result = Mid_16;
```

In this example the accumulator was loaded with the raw A/D value and then the offset was applied. The gain_factor was “pre-multiplied” by 256 (8 bits), giving it a granularity of 1/256. The result was extracted from the “middle” 16 bits of the accumulator (bits 8 to 23) to account for the scaling factor of 256 introduced in the multiply step.

The package requires about 384 bytes of ROM and 30 bytes of RAM. Individual routines can be deleted to conserve RAM if they are not used.
CODE SOURCE LISTINGS

NAME Math_32_Module

PUBLIC Load_16, ?Load_16?byte
PUBLIC Load_32, ?Load_32?byte
PUBLIC Mul_16, ?Mul_16?byte
PUBLIC Div_16, ?Div_16?byte
PUBLIC Add_16, ?Add_16?byte
PUBLIC Sub_16, ?Sub_16?byte
PUBLIC Add_32, ?Add_32?byte
PUBLIC Sub_32, ?Sub_32?byte
PUBLIC Low_16, Mid_16, High_16

Math_32_Data SEGMENT DATA
Math_32_Code SEGMENT CODE

RSEG Math_32_Data
?Load_16?byte: DS 2
?Load_32?byte: DS 4
?Mul_16?byte: DS 2
?Div_16?byte: DS 2
?Add_16?byte: DS 2
?Sub_16?byte: DS 2
?Add_32?byte: DS 4
?Sub_32?byte: DS 4
OP 0: DS 1
OP 1: DS 1
OP 2: DS 1
OP 3: DS 1
TMP 0: DS 1
TMP 1: DS 1
TMP 2: DS 1
TMP 3: DS 1

RSEG Math_32_Code
Load_16:
;Load the lower 16 bits of the OP registers with the value supplied
MOV OP_3,#0
MOV OP_2,#0
MOV OP_1,?Load_16?byte
MOV OP_0,?Load_16?byte + 1
RET

Load_32:
;Load all the OP registers with the value supplied
MOV OP_3,?Load_32?byte
MOV OP_2,?Load_32?byte + 1
MOV OP_1,?Load_32?byte + 2
MOV OP_0,?Load_32?byte + 3
RET

Low_16:
;Return the lower 16 bits of the OP registers
MOV R6,OP_1
MOV R7,OP_0
RET

Mid_16:
;Return the middle 16 bits of the OP registers
MOV R6,OP_2
MOV R7,OP_1
RET

High_16:
;Return the high 16 bits of the OP registers
MOV R6,OP_3
MOV R7,OP_2
RET

Add_16:
;Add the 16 bits supplied by the caller to the OP registers
CLR C
MOV A,OP_0
ADDC A,?Add_16?byte + 1 ;low byte first
MOV OP_0,A
MOV A,OP_1
ADDC A,?Add_16?byte ;high byte + carry
MOV OP_1,A
MOV A,OP_2
ADDC A,#0 ;propagate carry only
MOV OP_2,A
MOV A,OP_3
ADDC A,#0 ;propagate carry only
MOV OP_3,A
RET
Add_32:
;Add the 32 bits supplied by the caller to the OP registers
CLR C
MOV A,OP_0
ADDCC A,?Add_32?byte + 3 ;lowest byte first
MOV OP_0,A
MOV A,OP_1
ADDCC A,?Add_32?byte + 2 ;mid-lowest byte + carry
MOV OP_1,A
MOV A,OP_2
ADDCC A,?Add_32?byte + 1 ;mid-highest byte + carry
MOV OP_2,A
MOV A,OP_3
ADDCC A,?Add_32?byte ;highest byte + carry
MOV OP_3,A
RET

Sub_16:
;Subtract the 16 bits supplied by the caller from the OP registers
CLR C
MOV A,OP_0
SUBB A,?Sub_16?byte + 1 ;low byte first
MOV OP_0,A
MOV A,OP_1
SUBB A,?Sub_16?byte ;high byte + carry
MOV OP_1,A
MOV A,OP_2
SUBB A,#0 ;propagate carry only
MOV OP_2,A
MOV A,OP_3
SUBB A,#0 ;propagate carry only
MOV OP_3,A
RET

Sub_32:
;Subtract the 32 bits supplied by the caller from the OP registers
CLR C
MOV A,OP_0
SUBB A,?Sub_32?byte + 3 ;lowest byte first
MOV OP_0,A
MOV A,OP_1
SUBB A,?Sub_32?byte + 2 ;mid-lowest byte + carry
MOV OP_1,A
MOV A,OP_2
SUBB A,?Sub_32?byte + 1 ;mid-highest byte + carry
MOV OP_2,A
MOV A,OP_3
SUBB A,?Sub_32?byte ;highest byte + carry
MOV OP_3,A
RET
Mul_16:
; Multiply the 32 bit OP with the 16 value supplied
MOV TMP_3,#0 ; clear out upper 16 bits
MOV TMP_2,#0
; Generate the lowest byte of the result
MOV B,OP.0
MOV A,?Mul_16?byte+1
MUL AB
MOV TMP_0,A ; low-order result
MOV TMP_1,B ; high-order result
; Now generate the next higher order byte
MOV B,OP.1
MOV A,?Mul_16?byte+1
MUL AB
ADD A, TMP_1 ; low-order result
MOV TMP_1,A ; save
MOV A,B ; get high-order result
ADDC A, TMP_2 ; include carry from previous operation
MOV TMP_2,A ; save
JNC Mul_loop1
INC TMP_3 ; propagate carry into TMP_3
Mul_loop1:
MOV B,OP.0
MOV A,?Mul_16?byte
MUL AB
ADD A, TMP_1 ; low-order result
MOV TMP_1,A ; save
MOV A,B ; get high-order result
ADDC A, TMP_2 ; include carry from previous operation
MOV TMP_2,A ; save
JNC Mul_loop2
INC TMP_3 ; propagate carry into TMP_3
Mul_loop2:
; Now start working on the 3rd byte
MOV B,OP.2
MOV A,?Mul_16?byte+1
MUL AB
ADD A, TMP_2 ; low-order result
MOV TMP_2,A ; save
MOV A,B ; get high-order result
ADDC A, TMP_3 ; include carry from previous operation
MOV TMP_3,A ; save
; Now the other half
MOV B,OP.1
MOV A,?Mul_16?byte
MUL AB
ADD A, TMP_2 ; low-order result
MOV TMP_2,A ; save
MOV A,B ; get high-order result
ADDC A, TMP_3 ; include carry from previous operation
MOV TMP_3,A ; save
; Now finish off the highest order byte
MOV B,OP.3
MOV A,?Mul_16?byte+1
MUL AB
ADD A, TMP_3 ; low-order result
MOV TMP_3, A ; save
; Forget about the high-order result, this is only 32 bit math!
MOV B, OP_2
MOV A, ?MUL_16?byte
MUL AB
ADD A, TMP_3 ; low-order result
MOV TMP_3, A ; save
; Now we are all done, move the TMP values back into OP
MOV OP_0, TMP_0
MOV OP_1, TMP_1
MOV OP_2, TMP_2
MOV OP_3, TMP_3
RET

270530-5
Div 16:
; This divides the 32 bit OP register by the value supplied

MOV R7,#0
MOV R6,#0
MOV TMP_0,#0
MOV TMP_1,#0
MOV TMP_2,#0
MOV TMP_3,#0
MOV R1, #Div 16?byte
MOV R0, #Div 16?byte+1
MOV R5, #12

; This begins the loop

Div loop:
CALL Shift_D
MOV A,R6
RLC A
MOV R6,A
MOV A,R7
RLC A
MOV R7,A

; now test to see if R7>R6 >= R1:R0
CLR C
MOV A,R7
SUBB A,R1
JC Cant

; subtract R1 from R7 to see if R1 < R7

; A = R7 - R1, carry set if R7 < R1

; at this point R7>R1 or R7=R1
JNZ Can sub

; if R7 = R1, test for R6=R0
CLR C
MOV A,R6
SUBB A,R0
JC Cant sub

; subtract the divisor from the partial remainder
CLR C
SUBB A,R6
MOV R6,A
SUBB A,R0
MOV A,R7
SUBB A,R1
MOV R7,A
SETB C

; shift a 1 into the quotient
JMP Quot

Cant sub:
; subtract the dividend and return MSB in C

; shift the dividend and return MSB in C
CALL Shift_D
MOV A,R6
RLC A
MOV R6,A
MOV A,R7
RLC A
MOV R7,A

; now test to see if R7>R6 >= R1:R0
CLR C
MOV A,R7
SUBB A,R1
JC Cant

; subtract R1 from R7 to see if R1 < R7

; A = R7 - R1, carry set if R7 < R1

; at this point R7>R1 or R7=R1
JNZ Can sub

; if R7 = R1, test for R6=R0
CLR C
MOV A,R6
SUBB A,R0
JC Cant sub

; subtract the divisor from the partial remainder
CLR C
SUBB A,R6
MOV R6,A
SUBB A,R0
MOV A,R7
SUBB A,R1
MOV R7,A
SETB C

; shift a 1 into the quotient
JMP Quot

Cant sub:
; subtract the dividend and return MSB in C

; shift the dividend and return MSB in C
CALL Shift_D
MOV A,R6
RLC A
MOV R6,A
MOV A,R7
RLC A
MOV R7,A

; now test to see if R7>R6 >= R1:R0
CLR C
MOV A,R7
SUBB A,R1
JC Cant

; subtract R1 from R7 to see if R1 < R7

; A = R7 - R1, carry set if R7 < R1

; at this point R7>R1 or R7=R1
JNZ Can sub

; if R7 = R1, test for R6=R0
CLR C
MOV A,R6
SUBB A,R0
JC Cant sub

; subtract the divisor from the partial remainder
CLR C
SUBB A,R6
MOV R6,A
SUBB A,R0
MOV A,R7
SUBB A,R1
MOV R7,A
SETB C

; shift a 1 into the quotient
JMP Quot

Cant sub:
; subtract the dividend and return MSB in C

; shift the dividend and return MSB in C
CALL Shift_D
MOV A,R6
RLC A
MOV R6,A
MOV A,R7
RLC A
MOV R7,A

; now test to see if R7>R6 >= R1:R0
CLR C
MOV A,R7
SUBB A,R1
JC Cant

; subtract R1 from R7 to see if R1 < R7

; A = R7 - R1, carry set if R7 < R1

; at this point R7>R1 or R7=R1
JNZ Can sub

; if R7 = R1, test for R6=R0
CLR C
MOV A,R6
SUBB A,R0
JC Cant sub

; subtract the divisor from the partial remainder
CLR C
SUBB A,R6
MOV R6,A
SUBB A,R0
MOV A,R7
SUBB A,R1
MOV R7,A
SETB C

; shift a 1 into the quotient
JMP Quot

Cant sub:
; subtract the dividend and return MSB in C

; shift the dividend and return MSB in C
CALL Shift_D
MOV A,R6
RLC A
MOV R6,A
MOV A,R7
RLC A
MOV R7,A

; now test to see if R7>R6 >= R1:R0
CLR C
MOV A,R7
SUBB A,R1
JC Cant

; subtract R1 from R7 to see if R1 < R7

; A = R7 - R1, carry set if R7 < R1

; at this point R7>R1 or R7=R1
JNZ Can sub

; if R7 = R1, test for R6=R0
CLR C
MOV A,R6
SUBB A,R0
JC Cant sub

; subtract the divisor from the partial remainder
CLR C
SUBB A,R6
MOV R6,A
SUBB A,R0
MOV A,R7
SUBB A,R1
MOV R7,A
SETB C

; shift a 1 into the quotient
MOV OP_2, TMP_2
MOV OP_3, TMP_3
RET

Shift D:
; shift the dividend one bit to the left and return the MSB in C
CLR C
MOV A, OP_0
RLC A
MOV OP_0, A
MOV A, OP_1
RLC A
MOV OP_1, A
MOV A, OP_2
RLC A
MOV OP_2, A
MOV A, OP_3
RLC A
MOV OP_3, A
RET

Shift Q:
; shift the quotient one bit to the left and shift the C into LSB
MOV A, TMP_0
RLC A
MOV TMP_0, A
MOV A, TMP_1
RLC A
MOV TMP_1, A
MOV A, TMP_2
RLC A
MOV TMP_2, A
MOV A, TMP_3
RLC A
MOV TMP_3, A
RET

END

270530-7
Designing a Mailbox Memory for Two 80C31 Microcontrollers Using EPLDs

K. WEIGL & J. STAHL
INTEL CORPORATION
MUNICH, GERMANY
INTRODUCTION

Very often, complex systems involve two or more microcontrollers to fulfill the requirements defined by a given objective. Since the nature of microcontrollers does not allow for easy dual-port memory design (no “READY” input; no “HOLD/HLDA” interface; port-oriented I/O etc.), design engineers are faced with the problem of interchanging information (data and status) between those microcontrollers. This application brief describes the design of a mailbox for exchanging information between two 80C31s, using a 5C060 H-EPLD as a “back-to-back” register, and a 5C031 H-EPLD as an arbitration vehicle to control the actions of the CPUs.

THE 5C060 MAILBOX

In this application, the 16 macrocells of the 5C060 are grouped into two sets of 8 so called “ROIF” (register output with input feedback) primitives to implement the two 8 bit bus interfaces needed. The grouping is done according to the following picture.

The 5C060 allows for independent clocking of 8 macrocells on each side of the chip, the two clock inputs are used to clock data from the microcontroller bus into the chip. To read the data written into the mailbox by one of the controllers, the RDA- (controller A is reading) or RDB- (controller B is reading) line must be pulled low by activating the read command (IRD). In order to avoid spurious read-cycles, the /RD commands from both microcontrollers are logically “ORed” together with an active high CS-signal (Chip Select) inside the 5C060. The CS-signal for both ports is derived from address line A15. Therefore, whenever A15 becomes a logic “1” (true), the mailbox is activated and ready to take or submit data.

Address range for the mailbox: F000 Hex to FFFF Hex (Upper 12 kbyte)
THE 5C031 “MAILBOX CONTROLLER”

To keep the two microcontrollers informed about the status of their mailbox, the 5C031 is programmed to supply the following signals to both controllers:

/OBFA: "OUTPUT BUFFER FULL" FOR MC A
/OBFB: "OUTPUT BUFFER FULL" FOR MC B
/IBEA: "INPUT BUFFER EMPTY" FOR MC A
/IBEB: "INPUT BUFFER EMPTY" FOR MC B
/INTA: INTERRUPT TO MC A
/INTB: INTERRUPT TO MC B

The next section will discuss the meanings of these signals in more detail.

Output Buffer Full: This flag is set whenever the controller writes into its own output buffer. The flag remains valid, until the second controller has read the data. The flag is automatically reset to its inactive state when this read cycle is accomplished.

NOTE:
Both controllers can access (read or write) the mailbox simultaneously.

Input Buffer Empty: This flag indicates that there is no message in the mailbox. The flag will become inactive as soon as one microcontroller places a message for the other one (or vice versa).

Interrupt: The 5C031 is programmed to supply interrupts to both microcontrollers involved, on one of the following events.

1. The /OBF flag of the opposite microcontroller becomes active; e.g. if controller A is placing a message for controller B, controller B receives an interrupt the same time as /OBFA becomes valid or vice versa.

2. The /IBE flag of the opposite microcontroller goes active, indicating that this controller has received the message; e.g. if controller B reads the message stored by controller A, its /IBE flag goes active and controller receives an interrupt indicating that the buffer is empty.

The signals described above are necessary to accomplish a secure handshake without overwriting messages accidentally. In addition to that, the 5C031 is issuing the actual write commands for the two register sets inside the 5C060. The /WRA and /WRB signals are results of logical “AND” functions between the appropriate CS- and /WR signals from the microcontrollers. Therefore, spurious write cycles are unlikely to happen.

NOTE:
This design can also be efficiently implemented in a single 5CBIC EPLD.

Example: /IBE remains “LOW” until microcontroller B places a message for controller A into the mailbox for A. /IBE will go “HIGH” as soon as controller B has accomplished its write cycle, and will not go “LOW” again until microcontroller A has read the message.
5C060 "BACK TO BACK REGISTER"
**5C060 REGISTER ADF**

**JUERG STAHL**  
**INTEL ZUERICH**  
March 27, 1986  
**5C031 MAILBOX MEMORY USING 5C060 / 5C031**  

**EXAMPLE .ADF**

------

**INPUTS:** WB81, CSA82, CSB814, nRDA811, nRDB823, WA813  
**OUTPUTS:** IOB7815, IOA7810, IOB681S, IOA6815, IOB5817, IOA588, IOB4818, IOA487, IOB3819, IOA386, IOB2820, IOA285, IOB1821, IOA184, IOB0822, IOA083

**NETWORK:**

IOB7, DB7 = ROIF (DA7, WAC, GND, GND, RDBC)  
IOA7, DA7 = ROIF (DB7, WBC, GND, GND, RDAC)

IOB6, DB6 = ROIF (DA6, WAC, GND, GND, RDBC)  
IOA6, DA6 = ROIF (DB6, WBC, GND, GND, RDAC)

IOB5, DB5 = ROIF (DA5, WAC, GND, GND, RDBC)  
IOA5, DA5 = ROIF (DB5, WBC, GND, GND, RDAC)

IOB4, DB4 = ROIF (DA4, WAC, GND, GND, RDBC)  
IOA4, DA4 = ROIF (DB4, WBC, GND, GND, RDAC)

IOB3, DB3 = ROIF (DA3, WAC, GND, GND, RDBC)  
IOA3, DA3 = ROIF (DB3, WBC, GND, GND, RDAC)

IOB2, DB2 = ROIF (DA2, WAC, GND, GND, RDBC)  
IOA2, DA2 = ROIF (DB2, WBC, GND, GND, RDAC)

IOB1, DB1 = ROIF (DA1, WAC, GND, GND, RDBC)  
IOA1, DA1 = ROIF (DB1, WBC, GND, GND, RDAC)

IOB0, DB0 = ROIF (DA0, WAC, GND, GND, RDBC)  
IOA0, DA0 = ROIF (DB0, WBC, GND, GND, RDAC)

WAC = INP (WA)  
RDBC = AND(CSBI, RDBI)

WBC = INP (WB)  
RDAC = AND(CSAI, RDAI)

CSBI = INP (CSB)  
nRDBI = INP(nRDB)

nRDAI = INP(nRDA)  
CSAI = INP(CSA)

RDAI = NOT(nRDAI)  
RDBI = NOT(nRDBI)

END$  

292016-4
5C060 REGISTER LEF

JUERG STAHL
INTEL ZURICH
March 27, 1986

**EXAMPLE.LEF**

80C31 MAILBOX MEMORY USING 5C060 / 5C031

1

5C060

LB Version 3.0, Baseline 17x, 9/26/85
LEF Version 1.0 Baseline 1.51 02 Feb 1987

PART: 5C060

INPUTS: WB01, CSA02, CSB014, nRDA011, nRDB023, WA013

OUTPUTS: IOB7015, IOA7010, IOB6016, IOA6009, IOB5017, IOA5008, IOB4018, IOA4007,
          IOB3019, IOA3006, IOB2020, IOA2005, IOB1021, IOA1004, IOB0022, IOA003

NETWORK:

WBC = INP(WB)
WAC = INP(WA)
CSAI = INP(CSA)
CSBI = INP(CSB)
nRDAI = INP(nRDA)
nRDBI = INP(nRDB)

IOB7, DB7 = ROIF(DA7, WAC, GND, GND, RDBC)
IOA7, DA7 = ROIF(DB7, WBC, GND, GND, RDAC)
IOB6, DB6 = ROIF(DA6, WAC, GND, GND, RDBC)
IOA6, DA6 = ROIF(DB6, WBC, GND, GND, RDAC)
IOB5, DB5 = ROIF(DA5, WAC, GND, GND, RDBC)
IOA5, DA5 = ROIF(DB5, WBC, GND, GND, RDAC)
IOB4, DB4 = ROIF(DA4, WAC, GND, GND, RDBC)
IOA4, DA4 = ROIF(DB4, WBC, GND, GND, RDAC)
IOB3, DB3 = ROIF(DA3, WAC, GND, GND, RDBC)
IOA3, DA3 = ROIF(DB3, WBC, GND, GND, RDAC)
IOB2, DB2 = ROIF(DA2, WAC, GND, GND, RDBC)
IOA2, DA2 = ROIF(DB2, WBC, GND, GND, RDAC)
IOB1, DB1 = ROIF(DA1, WAC, GND, GND, RDBC)
IOA1, DA1 = ROIF(DB1, WBC, GND, GND, RDAC)
IOB0, DB0 = ROIF(DA0, WAC, GND, GND, RDBC)
IOA0, DA0 = ROIF(DB0, WBC, GND, GND, RDAC)

EQUATIONS:

RDAC = CSAI * nRDAI';
RDAC = CSBI * nRDBI';

END$
### Logic Optimizing Compiler Utilization Report

**FIT Version 1.0 Baseline 1.0i 2/6/87**

***** Design implemented successfully *****

**JUERG STAHL**
**INTEL ZUEICH**
March 27, 1986

**80C31 MAILBOX MEMORY USING 5C060 / 5C031**

**5C060**

**LB Version 3.0, Baseline 17x, 9/26/85**

```plaintext
5C060
  - - - -
  WB: 1  24: - Vcc
  CSA: 2  23: - nRDB
  IOA0: 3  22: - IOB0
  IOA1: 4  21: - IOB1
  IOA2: 5  20: - IOB2
  IOA3: 6  19: - IOB3
  IOA4: 7  18: - IOB4
  IOA5: 8  17: - IOB5
  IOA6: 9  16: - IOB6
  IOA7: 10  15: - IOB7
  nRDA: 11  14: - CSA
  GND: 12  13: - WA
  - - - -
```

**INPUTS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Pin</th>
<th>Resource</th>
<th>MCell #</th>
<th>PTerms</th>
<th>Feeds:</th>
<th>MCells</th>
<th>OE</th>
<th>Clear</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>WB</td>
<td>1</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSA</td>
<td>2</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nRDA</td>
<td>11</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GND</td>
<td>12</td>
<td>GND</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

292016-6
### 5C060 REGISTER UTILIZATION REPORT (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Pin</th>
<th>Resource</th>
<th>MCel #</th>
<th>PTerm</th>
<th>MCells</th>
<th>OE</th>
<th>Clear</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOA0</td>
<td>3</td>
<td>ROIF</td>
<td>9</td>
<td>1/8</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA1</td>
<td>4</td>
<td>ROIF</td>
<td>10</td>
<td>1/8</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA2</td>
<td>5</td>
<td>ROIF</td>
<td>11</td>
<td>1/8</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA3</td>
<td>6</td>
<td>ROIF</td>
<td>12</td>
<td>1/8</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA4</td>
<td>7</td>
<td>ROIF</td>
<td>13</td>
<td>1/8</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA5</td>
<td>8</td>
<td>ROIF</td>
<td>14</td>
<td>1/8</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA6</td>
<td>9</td>
<td>ROIF</td>
<td>15</td>
<td>1/8</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA7</td>
<td>10</td>
<td>ROIF</td>
<td>16</td>
<td>1/8</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA8</td>
<td>11</td>
<td>ROIF</td>
<td>17</td>
<td>1/8</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA9</td>
<td>12</td>
<td>ROIF</td>
<td>18</td>
<td>1/8</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA10</td>
<td>13</td>
<td>ROIF</td>
<td>19</td>
<td>1/8</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA11</td>
<td>14</td>
<td>ROIF</td>
<td>20</td>
<td>1/8</td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IOA12</td>
<td>15</td>
<td>ROIF</td>
<td>21</td>
<td>1/8</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**All Resources used**

**PART UTILIZATION**

- 100% Pins
- 100% MacroCells
- 12% Pterms
5C031 ARBITER ADF

JUERG STAHL
INTEL ZURICH
March 28, 1986

5C031 MAILBOX MEMORY USING 5C060 / 5C031

PART: 5C031

** EXAMPLE .ADF **

INPUTS: RST, nWRA, nRDB, CSA, nRDA, nWRB, CSB, nOE
OUTPUTS: WA, nOBFA, nIBEB, nINTA, nINTB, nOBFB, nIBEA, WB

NETWORK:

nWRA = INP(nWRA)
nRDB = INP(nRDB)
RST = INP(RST)
CSA = INP(CSA)
nRDA = INP(nRDA)
nWRB = INP(nWRB)
CSB = INP(CSB)
nOE = INP(nOE)
WRA = NOT(nWRA)
WB = NOT(nWRB)
RDA = NOT(nRDA)
RDB = NOT(nRDB)
OE = NOT(nOE)

nRST = NOT(RST)
WA = CONF(WAd, YCC)
WAd = AND(CSA, WRA)
WB = CONF(WBd, YCC)
WBd = AND(CSB, WB)
nRB = NAND(RDB, CSB)
nRA = NAND(RDA, CSA)
nWAd = NOT(WAd)
nWBd = NOT(WBd)
nOBFA, nOBFA = COCF(nOBFAd, OE)
nOBFB, nOBFB = COCF(nOBFBd, OE)
nIBEB, nIBEB = COCF(nIBEAd, OE)
nIBEB, nIBEB = COCF(nIBEBd, OE)
nINTA = CONF(nINTAd, OE)
nINTad = AND(nOBFA, nIBEA)
nINTbd = AND(nOBFB, nIBEB)
nOBFAd = NAND(nRA, nIBEB, nRST)
nOBFAd = NAND(nWBd, nOBFA)
nIBEAd = NAND(nWBd, nOBFB)

ENDS
5C031 ARBITER LEF

JUERG STAHL
INTEL ZUERICH
March 28, 1986

80C31 MAILBOX MEMORY USING 5C060 / 5C031

LB Version 3.0, Baseline 17x, 9/26/85
LEF Version 1.0 Baseline 1.5i 02 Feb 1987

PART: 5C031

INPUTS: RST, nWRA, nRDB, CSA, nRDA, nWRB, CSB, nOE

OUTPUTS: WA, nOBFA, nIBEB, nINTA, nINTB, nOBFB, nIBEA, WB

NETWORK:

RST = INP(RST)
nWRA = INP(nWRA)
nRDB = INP(nRDB)
CSA = INP(CSA)
nRDA = INP(nRDA)
nWRB = INP(nWRB)
CSB = INP(CSB)
nOE = INP(nOE)
WA = CONF(WAd, VCC)
nOBFA, nOBFB = COCF(nOBFAd, OE)
nIBEB, nIBEA = COCF(nIBEBd, OE)
nINTA = CONF(nINTAd, OE)
nINTB = CONF(nINTBd, OE)
nOBFB, nOBFB = COCF(nOBFBd, OE)
nIBEA, nIBEB = COCF(nIBEAd, OE)
WB = CONF(WBd, VCC)

EQUATIONS:

WBd = CSB * nWRB';
nIBEAd = CSB * nWRB' + nOBFB';
nOBFBd = (nIBEA * RST' * CSA' + nIBEA * RST' * nRDA)';
nINTBd = nOBFB * nIBEB;
nINTAd = nOBFA * nIBEA;
nIBEBd = CSA * nWRA' + nOBFA';
OE = nOE';
nOBFAd = (nIBEB * RST' * CSB' + nIBEB * RST' * nRDB)';
WAd = CSA * nWRA';

END$
## 5C031 ARBITER LEF (Continued)

Logic Optimizing Compiler Utilization Report  
FIT Version 1.0 Baseline 1.0i 2/6/87

****** Design implemented successfully

JUERG STAHL  
INTEL ZUERICH  
March 28, 1986

80C31 MAILBOX MEMORY USING 5C060 / 5C031  
2  
5C031

LB Version 3.0, Baseline 17x, 9/26/85  
5C031

<table>
<thead>
<tr>
<th>Name</th>
<th>Pin</th>
<th>Resource</th>
<th>MCell #</th>
<th>PTerms</th>
<th>MCells</th>
<th>OE</th>
<th>Clear</th>
<th>Preset</th>
</tr>
</thead>
<tbody>
<tr>
<td>nOE</td>
<td>3</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSB</td>
<td>4</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nWRB</td>
<td>5</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nRDA</td>
<td>6</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSA</td>
<td>7</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nRDB</td>
<td>8</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nWRA</td>
<td>9</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GND</td>
<td>10</td>
<td>GND</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RST</td>
<td>11</td>
<td>INP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Vcc</td>
<td>20</td>
<td>Vcc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

292016-11
## 5C031 Arbiter Utilization Report

### **Outputs**

<table>
<thead>
<tr>
<th>Name</th>
<th>Pin</th>
<th>Resource</th>
<th>MCell</th>
<th>PTerms</th>
<th>MCells</th>
<th>OK</th>
<th>Clear</th>
<th>Preset</th>
</tr>
</thead>
<tbody>
<tr>
<td>nIBEA</td>
<td>12</td>
<td>COCF</td>
<td>8</td>
<td>2/8</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nOBFA</td>
<td>13</td>
<td>COCF</td>
<td>7</td>
<td>2/8</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nIBEB</td>
<td>14</td>
<td>COCF</td>
<td>6</td>
<td>2/8</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nINTA</td>
<td>15</td>
<td>CONF</td>
<td>5</td>
<td>1/8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nINTB</td>
<td>16</td>
<td>CONF</td>
<td>4</td>
<td>1/8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nOBFB</td>
<td>17</td>
<td>COCF</td>
<td>3</td>
<td>2/8</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WA</td>
<td>18</td>
<td>CONF</td>
<td>2</td>
<td>1/8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WB</td>
<td>19</td>
<td>CONF</td>
<td>1</td>
<td>1/8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Unused Resources

<table>
<thead>
<tr>
<th>Name</th>
<th>Pin</th>
<th>Resource</th>
<th>MCell</th>
<th>PTerms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Part Utilization

- **88%** Pins
- **100%** MacroCells
- **18%** Pterms
Designing With The 80C51BH

TOM WILLIAMSON
MCO APPLICATIONS ENGINEER
CMOS EVOLVES

The original CMOS logic families were the 4000-series and the 74C-series circuits. The 74C-series circuits are functional equivalents to the corresponding numbered 74-series TTL circuits, but have CMOS logic levels and retain the other well known characteristics of CMOS logic.

These characteristics are: low power consumption, high noise immunity, and slow speed. The low power consumption is inherent to the nature of the CMOS circuit. The noise immunity is due partly to the CMOS logic levels, and partly to the slowness of the circuits. The slow speed is due to the technology used to construct the transistors in the circuit.

The technology used is called metal-gate CMOS, because the transistor gates are formed by metal deposition. More importantly, the gates are formed after the drain and source regions have been defined, and must overlap the source and drain somewhat to allow for alignment tolerances. This overlap plus the relatively large size of the transistors themselves result in high electrode capacitance, and that is what limits the speed of the circuit.

High speed CMOS became feasible with the development of the self-aligning silicon gate technology. In this process polysilicon gates are deposited before the source and drain regions are defined. Then the source and drain regions are formed by ion implantation using the gate itself as a mask for the implantation. This eliminates most of the overlap capacitance. In addition, the process allows smaller transistors. The result is a significant increase in circuit speed. The 74HC-series of CMOS logic circuits is based on this technology, and has speeds comparable to LS TTL, which is to say about 10 times faster than the 74C-series circuits.

The size reduction that contributes to the higher speed also demands an accompanying reduction in the maximum supply voltage. High-speed CMOS is generally limited to 6V.

WHAT IS CHMOS?

CHMOS is the name given to Intel's high-speed CMOS processes. There are two CHMOS processes, one based on an n-well structure and one based on a p-well structure. In the n-well structure, n-type wells are diffused into a p-type substrate. Then the n-channel transistors (nFETs) are built into the substrate and pFETs are built into the n-wells. In the p-well structure, p-type wells are diffused into an n-type substrate. Then the nFETs are built into the wells and pFETs, into the substrate. Both processes have their advantages and disadvantages, which are largely transparent to the user.

Lower operating voltages are easier to obtain with the p-well structure than with the n-well structure. But the p-well structure does not easily adapt to an EPROM which would be pin-for-pin compatible with HMOS EPROMs. On the other hand the n-well structure can be based on the solidly founded HMOS process, in which nFETs are built into a p-type substrate. This allows somewhat more than half of the transistors in a CHMOS chip to be constructed by processes that are already well characterized.

Currently Intel's CHMOS microcontrollers and memory products are n-well devices, whereas CHMOS microprocessors are p-well devices.

Further discussion of the CHMOS technology is provided in References 1 and 2 (which are reprinted in the Microcontroller Handbook).

THE MCS®-51 FAMILY IN CHMOS

The 80C51BH is the CHMOS version of Intel's original 8051. The 80C31BH is the ROMless 80C51BH, equivalent to the 8031. These CHMOS devices are architecturally identical with their HMOS counterparts, except that they have two added features for reduced power. These are the Idle and Power Down modes of operation.

In most cases, an 80C51BH can directly replace the 8051 in existing applications. It can execute the same code at the same speed, accept signals from the same sources, and drive the same loads. However, the 80C51BH covers a wider range of speeds, will emit CMOS logic levels to CMOS loads, and will draw about 1/10 the current of an 8051 (and less yet in the reduced power modes). Interchangeability between the HMOS and CHMOS devices is discussed in more detail in the final section of this Application Note.

It should be noted that the 80C51BH CPU is not static. That means if the clock frequency is too low, the CPU might forget what it was doing. This is because the circuitry uses a number of dynamic nodes. A dynamic node is one that uses the note-to-ground capacitance to form a temporary storage cell. Dynamic nodes are used to reduce the transistor count, and hence the chip area, thus to produce a more economical device.

This is not to say that the on-chip RAM in CHMOS microcontrollers is dynamic. It's not. It's the CPU that is dynamic, and that is what imposes the minimum clock frequency specification.
LATCHUP

Latchup is an SCR-type turn-on phenomenon that is the traditional nemesis of CMOS systems. The substrate, the wells, and the transistors form parasitic pnpn structures within the device. These parasitic structures turn on like an SCR if a sufficient amount of forward current is driven through one of the junctions. From the circuit designer’s point of view it can happen whenever an input or output pin is externally driven a diode drop above $V_{CC}$ or below $V_{SS}$, by a source that is capable of supplying the required trigger current.

However much of a problem latchup has been in the past, it is good to know that in most recently developed CMOS devices, and specifically in CHMOS devices, the current required to trigger latchup is typically well over 100 mA. The 80C51BH is virtually immune to latchup. (References 1 and 2 present a discussion of the latchup mechanisms and the steps that are taken on the chip to guard against it.) Modern CMOS is not absolutely immune to latchup, but with trigger currents in the hundreds of mA, latchup is certainly a lot easier to avoid than it once was.

A careless power-up sequence might trigger a latchup in the older CMOS families, but it’s unlikely to be a major problem in high-speed CMOS or in CHMOS. There is still some risk incurred in inserting or removing chips or boards in a CMOS system while the power is on. Also, severe transients, such as inductive kicks or momentary short-circuits, can exceed the trigger current for latchup.

For applications in which some latchup risk seems unavoidable, you can put a small resistor (100Ω or so) in series with signal lines to ensure that the trigger current will never be reached. This also helps to control overshoot and RFI.

LOGIC LEVELS AND INTERFACE PROBLEMS

CMOS logic levels differ from TTL levels in two ways. First, for equal supply voltages, CMOS gives (and requires) a higher “logic 1” level than TTL. Secondly, CMOS logic levels are $V_{CC}$ (or VDD) dependent, whereas guaranteed TTL logic levels are fixed when $V_{CC}$ is within TTL specs.

Standard 74HC logic levels are as follows:

- $V_{IHMIN} = 70\%$ of $V_{CC}$
- $V_{ILMAX} = 20\%$ of $V_{CC}$
- $V_{OHHMIN} = V_{CC} - 0.1V$, $|I_{OH}| \leq 20 \mu A$
- $V_{OLMAX} = 0.1V$, $|I_{OL}| \leq 20 \mu A$

Figure 1 compares 74HC, LS TTL, and 74HCT logic levels with those of the HMOS 8051 and the CHMOS 80C51BH for $V_{CC} = 5V$.

Output logic levels depend of course on load current, and are normally specified at several load currents. When CMOS and TTL are powered by the same $V_{CC}$, the logic levels guaranteed on the data sheets indicate that CMOS can drive TTL, but TTL can’t drive CMOS. The incompatibility is that the TTL circuit’s $V_{OH}$ level is too low to reliably be recognized by the CMOS circuit as a valid $V_{IH}$.

Since HMOS circuits were designed to be TTL-compatible, they have the same incompatibility.

Fortunately, 74HCT-series circuits are available to ease these interfacing problems. They have TTL-compatible logic levels at the inputs and standard CMOS levels at the outputs.

The 80C51BH is designed to work with either TTL or CMOS. Therefore its logic levels are specified very much like 74HCT circuits. That is, its input logic levels are TTL-compatible, and its output characteristics are like standard high-speed CMOS.

NOISE CONSIDERATIONS

One of the major reasons for going to CMOS has traditionally been that CMOS is less susceptible to noise. As previously noted, its low susceptibility to noise is
partly due to superior noise margins, and partly due to its slow speed.

Noise margin is the difference between $V_OH$ and $V_{IL}$, or between $V_{OH}$ and $V_{IH}$. If $V_{OH}$ from a driving circuit is 2.7V and $V_{IH}$ to the driven circuit is 2.0V, then the driven circuit has 0.7V of noise margin at the logic high level. These kinds of comparisons show that an all-CMOS system has wider noise margins than an all-TTL system.

Figure 2 shows noise margins in CMOS and LS TTL systems when both have $V_{CC} = 5V$. It can be seen that CMOS/CMOS and CMOS/CHMOS systems have an edge over LS TTL in this respect.

Noise margins can be misleading, however, because they don't say how much noise energy it takes to induce a logic error. This would involve consideration of the width of the noise pulse as compared with the circuit's response speed, and the impedance to ground from the point of noise introduction in the circuit.

When these considerations are included, it is seen that using the slower 74C- and 4000-series circuits with a 12 or 15V supply voltage does offer a truly improved level of noise immunity, but that high-speed CMOS at 5V is not significantly better than TTL.

One should not mistake the wider supply voltage tolerance of high-speed CMOS for $V_{CC}$ glitch immunity. Supply voltage tolerance is a DC rating, not a glitch rating.

For any clocked CMOS, and most especially for VLSI CMOS, $V_{CC}$ decoupling is critical. CHMOS draws current in extremely sharp spikes at the clock edges. The VHF and UHF components of these spikes are not drawn from the power supply, but from the decoupling capacitor. If the decoupling circuit is not sufficiently low in inductance, $V_{CC}$ will glitch at each clock edge. We suggest that a 0.1 μF decoupler cap be used in a minimum-inductance configuration with the microcontroller. A minimum-inductance configuration is one that minimizes the area of the loop formed by the chip ($V_{CC}$ to $V_{SS}$), the traces to the decoupler cap, and the decoupler cap. PCB designers too often fail to understand that if the traces that connect the decoupler cap to the $V_{CC}$ and $V_{SS}$ pins aren't short and direct, the decoupler loses much of its effectiveness.

Overshoot and ringing in signal lines are potential sources of logic upsets. These can largely be controlled by circuit layout. Inserting small resistors (about 100Ω) in series with signal lines that seem to need them will also help.

The sharp edges produced by high-speed CMOS can cause RFI problems. The severity of these problems is largely a function of the PCB layout. We don't mean to imply that all RFI problems can be solved by a better PCB layout. It may well be, for example, that in some RFI-sensitive designs high-speed CMOS is simply not the answer. But circuit layout is a critical factor in the noise performance of any electronic system, and more so in high-speed CMOS systems than others.

Circuit layout techniques for minimizing noise susceptibility and generation are discussed in References 3 through 6.

### UNUSED PINS

CMOS input pins should not be left to float, but should always be pulled to one logic level or the other. If they float, they tend to float into the transition region between 0 and 1, where the pullup and pulldown devices in the input buffer are both conductive. This causes a significant increase in $I_{CC}$. A similar effect exists in HMOS circuits, but with less noticeable results.

In 80C51BH and 80C31BH designs, unused pins of Ports 1, 2, and 3 can be ignored, because they have internal pullups that will hold them at a valid Logic 1 level. Port 0 pins are different, however, in not having internal pullups (except during bus operations).

When the 80C51BH is in reset, the Port 0 pins are in a float state unless they are externally pulled up or down. If it's going to be held in reset for just a short time, the transient float state can probably be ignored. When it comes out of reset, the pins stay afloat unless

<table>
<thead>
<tr>
<th>Interface</th>
<th>Noise Margin for $V_{CC} = 5V$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Logic Low $V_{IL}$ - $V_{OL}$</td>
</tr>
<tr>
<td>74HC to 74HC</td>
<td>0.9V</td>
</tr>
<tr>
<td>LS TTL to LS TTL</td>
<td>0.3V</td>
</tr>
<tr>
<td>LS TTL to 74HCT</td>
<td>0.3V</td>
</tr>
<tr>
<td>LS TTL to 80C51BH</td>
<td>0.3V</td>
</tr>
<tr>
<td>74HC to 80C51BH</td>
<td>0.8V</td>
</tr>
<tr>
<td>80C51BH to 74HC</td>
<td>0.8V</td>
</tr>
</tbody>
</table>

**Figure 2. Noise Margins for CMOS and LS TTL Circuits**

2-192
they are externally pulled either up or down. Alternatively, the software can internally write 0s to whatever Port 0 pins may be unused.

The same considerations are applicable to the 80C31BH with regards to reset. But when the 80C31BH comes out of reset, it commences bus operations, during which the logic levels at the pins are always well defined as high or low.

Consider the 80C31BH in the Power Down or Idle modes, however. In those modes it is not fetching instructions, and the Port 0 pins will float if not externally pulled high or low. The choice of whether to pull them high or low is the designer's. Normally it is sufficient to pull them up to VCC with 10k resistors. But if power is going to be removed from circuits that are connected to the bus, it will be advisable to pull the bus pins down (normally with 10k resistors). Considerations involved in selecting pullup and pulldown resistor values are as follows.

**Pullup Resistors**

If a pullup resistor is to be used on a Port 0 pin, its minimum value is determined by IOH requirements. If the pin is trying to emit a 0, then it will have to sink the current from the pullup resistor plus whatever other current may be sourced by other loads connected to the pin, as shown in Figure 3a. To guarantee that the pin voltage will not exceed 0.45V, the resistor should be selected so that IOH doesn't exceed the value specified on the data sheet. In most CMOS applications, the minimum value would be about 2k Ω.

The maximum value you could use depends on how fast you want the pin to pull up after bus operations have ceased, and how high you want the VOH level to be. The smaller the resistor the faster it pulls up. Its effect on the VOH level is that 

\[
VOH = VCC - (ILI + IIH) \times R
\]

where $ILI$ is the input leakage current to the Port 0 pin, and $IIH$ is the input high current to the external loads, as shown in Figure 3b. Normally $VOH$ can be expected to reach 0.9 VCC if the pullup resistance does not exceed about 50k Ω.

**Pulldown Resistors**

If a pulldown resistor is to be used on a Port 0 pin, its minimum value is determined by VOH requirements during bus operations, and its maximum value is in most cases determined by leakage current.

During bus operations the port uses internal pullups to emit 1s. The D.C. Characteristics in the data sheet list guaranteed VOH levels for given IOH currents. (The "−" sign in the IOH value means the pin is sourcing that current to the external load, as shown in Figure 4.) To ensure the VOH level listed in the data sheet, the resis-
tor has to satisfy where $I_{IH}$ is the input high current to the external loads.

When the pin goes into a high impedance state, the pulldown resistor will have to sink leakage current from the pin, plus whatever other current may be sourced by other loads connected to the pin, as shown in Figure 4b. The Port 0 leakage current is $I_{IL}$ on the data sheet. The resistor should be selected so that the voltage developed across it by these currents will be seen as a logic low by whatever circuits are connected to it (including the 80C51BH). In CMOS/CHMOS applications, 50k Ω is normally a reasonable maximum value.

\[ \frac{V_{OH}}{R} + I_{IH} \leq I_{OH} \]

Figure 4b. Conditions defining the maximum value for $R$. P0.X is in a high impedance state. $R$ must be small enough to keep $V_{OL}$ acceptably low.

**DRIVE CAPABILITY OF THE INTERNAL PULLUPS**

There's an important difference between HMOS and CHMOS port drivers. The pins of Ports 1, 2, and 3 of the CHMOS parts each have three pullups: strong, normal, and weak, as shown in Figure 5. The strong pullup (p1) is only used during 0-to-1 transitions, to hasten the transition. The weak pullup (p2) is on whenever the bit latch contains a 1. The "normal" pullup (p3) is controlled by the pin voltage itself.

The reason that p3 is controlled by the pin voltage is that if the pin is being used as an input, and the external source pulls it to a low, then turning off p3 makes for a lower $I_{IL}$. The data sheet shows an "ITL" specification. This is the current that p3 will source during the time the pin voltage is making its 1-to-0 transition. This is what $I_{IL}$ would be if an input low at the pin didn't turn p3 off.

Note, however, that this p3 turn-off mechanism puts a restriction on the drive capacity of the pin if it's being used as an output. If you're trying to output a logic high, and the external load pulls the pin voltage below the pin's $V_{IH\text{MIN}}$ spec, p3 might turn off, leaving only the weak p2 to provide drive to the load. To prevent this happening, you need to ensure that the load doesn't draw more than the $I_{OH}$ spec for a valid $V_{OH}$. The idea is to make sure the pin voltage never falls below its own $V_{IH\text{MIN}}$ specification.

![Figure 5. 80C51BH Output Drivers for Ports 1, 2 and 3](image)
POWER CONSUMPTION

The main reason for going to CMOS, of course, is to conserve power. (There are other reasons, but this is the main one.) Conserving power doesn't mean just reducing your electric bill. Nor does it necessarily relate to battery operation, although battery operation without CMOS is pretty unhandy. The main reason for conserving power is to be able to put more functionality into a smaller space. The reduced power consumption allows the use of smaller and lighter power supplies, and less heat being generated allows denser packaging of circuit components. Expensive fans and blowers can usually be eliminated.

A cooler running chip is also more reliable, since most random and wearout failures relate to die temperature. And finally, the lower power dissipation will allow more functions to be integrated onto the chip.

The reason CMOS consumes less power than NMOS is that when it's in a stable state there is no path of conduction from $V_{EE}$ to $V_{SS}$ except through various leakage paths. CMOS does draw current when it's changing states. How much current it draws depends on how often and how quickly it changes states.

![Figure 6. 80C51BH ICC vs. Clock Frequency](270068-6)

CMOS circuits draw current in sharp spikes during logical transitions. These current spikes are made up of two components. One is the current that flows during the transition time when pullup and pulldown FETs are both active. The average (DC) value of this component is larger when the transition times of the input signals are longer. For this reason, if the current draw is a critical factor in the design, slow rise and fall times should be avoided, even when the system speed doesn't seem to justify a need for nanosecond switching speeds.

The other component is the current that charges stray and load capacitance at the nodes of a CMOS logic gate. The average value of this current spike is its area (integral over time) multiplied by its rep rate. Its area is the amount of charge it takes to raise the node capacitance, $C$, to $V_{CC}$. That amount of charge is just $C \times V_{CC}$. So the average value of the current spike is $C \times V_{CC} \times f$, where $f$ is the clock frequency.

This component of current increases linearly with clock frequency. For minimal current draw, the 80C52BH-2 is spec'd to run at frequencies as low as 500 kHz.

Keep in mind, though, that other component of current that is due to slow rise and fall times. A sinusoid is not the optimal waveform to drive the XTAL1 pin with. Yet crystal oscillators, including the one on the 80C51BH, generate sinusoidal waveforms. Therefore, if the on-chip oscillator is being used, you can expect the device to draw more current at 500 kHz, than it does at 1.5 MHz, as shown in Figure 6. If you derive a good sharp square wave from an external oscillator, and use that to drive XTAL1, then the microcontroller will draw less current. But the external oscillator will probably make up the difference.

The 80C51BH has two power-saving features not available in the HMOS devices. These are the Idle and Power Down modes of operation. The on-chip hardware that implements these reduced power modes is shown in Figure 7. Both modes are invoked by software.

![Figure 7. Oscillator and Clock Circuitry Showing Idle and Power Down Hardware](270068-7)
Idle: In the Idle Mode (\texttt{IDL} = 0 in Figure 7), the CPU puts itself to sleep by gating off its own clock. It doesn’t stop the oscillator. It just stops the internal clock signal from getting to the CPU. Since the CPU draws 80 to 90 percent of the chip’s power, shutting it off represents a fairly significant power savings. The on-chip peripherals (timers, serial port, interrupts, etc.) and RAM continue to function as normal. The CPU status is preserved in its entirety: the Stack Pointer, Program Counter, Program Status Word, Accumulator, and all other registers maintain their data during Idle.

The Idle Mode is invoked by setting bit 0 (IDL) of the \texttt{PCON} register. \texttt{PCON} is not bit-addressable, so the bit has to be set by a byte operation, such as

\texttt{ORL PCON, #1}

The \texttt{PCON} register also contains flag bits \texttt{GFO} and \texttt{GFl}, which can be used for any general purposes, or to give an indication if an interrupt occurred during normal operation or during Idle. In this application, the instruction that invokes Idle also sets one or both of the flag bits. Their status can then be checked in the interrupt routines.

While the device is in the Idle Mode, ALE and PSEN emit logic high (\texttt{VOH}), as shown in Figure 8. This is so external EPROM can be deselected and have its output disabled.

The port pins hold the logical states they had at the time the Idle was activated. If the device was executing out of external program memory, Port 0 is left in a high impedance state and Port 2 continues to emit the high byte of the program counter (using the strong pullups to emit 1s). If the device was executing out of internal program memory, Ports 0 and 2 continue to emit whatever is in the P0 and P2 registers.

There are two ways to terminate Idle. Activation of any enabled interrupt will cause the hardware to clear bit 0 of the \texttt{PCON} register, terminating the Idle mode. The interrupt will be serviced, and following RETI the next instruction to be executed will be the one following the instruction that invoked Idle.

The other way is with a hardware reset. Since the clock oscillator is still running, RST only needs to be held active for two machine cycles (24 oscillator periods) to complete the reset. Note that this exit from Idle writes 1s to all the ports, initializes all SFRs to their reset values, and restarts program execution from location 0.

Power Down: In the Power Down Mode (\texttt{PD} = 0 in Figure 7), the CPU puts the whole chip to sleep by turning off the oscillator. In case it was running from an external oscillator, it also gates off the path to the internal phase generators, so no internal clock is generated even if the external oscillator is still running. The on-chip RAM, however, saves its data, as long as \texttt{VCC} is maintained. In this mode the only \texttt{Icc} that flows is leakage, which is normally in the micro-amp range.

The Power Down Mode is invoked by setting bit 1 in the \texttt{PCON} register, using a byte instruction such as

\texttt{ORL PCON, #2}

While the device is in Power Down, ALE and PSEN emit lows (\texttt{VOL}), as shown in Figure 8. The reason they are designed to emit lows is so that power can be removed from the rest of the circuit, if desired, while the 80CS51BH is in its Power Down mode.

The port pins continue to emit whatever data was written to them. Note that Port 2 emits its P2 register data even if execution was from external program memory.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Internal Execution</th>
<th>External Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idle</td>
<td>Power Down</td>
</tr>
<tr>
<td>ALE</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PSEN/</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P0</td>
<td>SFR Data</td>
<td>SFR Data</td>
</tr>
<tr>
<td>P1</td>
<td>SFR Data</td>
<td>SFR Data</td>
</tr>
<tr>
<td>P2</td>
<td>SFR Data</td>
<td>SFR Data</td>
</tr>
<tr>
<td>P3</td>
<td>SFR Data</td>
<td>SFR Data</td>
</tr>
</tbody>
</table>

Figure 8. Status of Pins in Idle and Power Down Modes. "SFR data" means the port pins emit their internal register data. "PCH" is the high byte of the Program Counter.
Port 0 also emits its P0 register data, but if execution was from external program memory, the P0 register data is FF. The oscillator is stopped, and the part remains in this state as long as \( V_{CC} \) is held, and until it receives an external reset signal.

The only exit from Power Down is a hardware reset. Since the oscillator was stopped, RST must be held active long enough for the oscillator to re-start and stabilize. Then the reset function initializes all the Special Function Registers (ports, timers, etc.) to their reset values, and re-starts the program from location 0. Therefore, timer reloads, interrupt enables, baud rates, port status, etc. need to be re-established. Reset does not affect the content of the on-chip data RAM. If \( V_{CC} \) was held during Power Down, the RAM data is still good.

**USING THE POWER DOWN MODE**

The software-invoked Power Down feature offers a means of reducing the power consumption to a mere trickle in systems which are to remain dormant for some period of time, while retaining important data.

The user should give some thought to what state the port pins should be left in during the time the clock is stopped, and write those values to the port latches before invoking Power Down.

If \( V_{CC} \) is going to be held to the entire circuit, one would want to write values to the port latches that would deselect peripherals before invoking Power Down. For example, if external memory is being used, the P2 SFR should be loaded with a value which will not generate an active chip select to any memory device.

In some applications, \( V_{CC} \) to part of the system may be shut off during Power Down, so that even quiescent and standby currents are eliminated. Signal lines that connect to those chips must be brought to a logic low, whether the chip in question is CMOS, NMOS, or TTL, before \( V_{CC} \) is shut off to them. CMOS pins have parasitic pn junctions to \( V_{CC} \) which will be forward biased if \( V_{CC} \) is reduced to zero while the pin is held at a logic high. NMOS pins often have FETs that look like diodes to \( V_{CC} \). TTL circuits may actually be damaged by an input high if \( V_{CC} = 0 \). That's why the 80C51BH outputs lows at ALE and PSEN during Power Down.

Figure 9 shows a circuit that can be used to turn \( V_{CC} \) off to part of the system during Power Down. The circuit will ensure that the secondary circuit is not de-energized until after the 80C31BH is in Power Down, and that the 80C31BH does not receive a reset (terminating the Power Down mode) before the secondary circuit is re-energized. Therefore, the program memory itself can be part of the secondary circuit.

Figure 9. The 80C31BH de-energies part of the circuit (VCC2) when it goes into Power Down. Selections of R and Q2 depend on VCC2 current draw.
In Figure 9, when V_{CC} is switched on to the 80C31BH, capacitor C1 provides a power-on reset. The reset function writes 1s to all the port pins. The 1 at P2.6 turns Q1 on, enabling V_{CC} to the secondary circuit through transistor Q2. As the 80C31BH comes out of reset, Port 2 commences emitting the high byte of the Program Counter, which results in the P2.7 and P2.6 pins outputting Os. The 0 at P2.7 ensures continuation of V_{CC} to the secondary circuit.

The system software must now write a 1 to P2.7 and a 0 to P2.6 in the Port 2 SFR, P2. These values will not appear at the Port 2 pins as long as the device is fetching instructions from external program memory. However, whenever the 80C31BH goes into Power Down, these values will appear at the port pins, and will shut off both transistors, disabling V_{CC} to the secondary circuit.

Closing the switch S1 re-energizes the secondary circuit, and at the same time sends a reset through C2 to the 80C31BH to wake it up. The diode D1 is to prevent C1 from hogging current from C2 during this secondary reset. D2 prevents C2 from discharging through the RST pin when V_{CC} to the secondary circuit goes to zero.

**Figure 10. Using Power MOSFETs to Control Vcc**

Power MOSFETs are gaining in popularity (and availability). The easiest way to control V_{CC} is with a Logic Level pFET, as shown in Figure 10a. This circuit allows the full V_{CC} to be used to turn the device on. Unfortunately, power pFETs are not economically competitive with bipolar transistors of comparable ratings.

Power nFETs are both economical and available, and can be used in this application if a DC supply of higher voltage is available to drive the gate. Figure 10b shows how to implement a V_{CC} switch using a power nFET and a (nominally) +12V supply. The problem here is that if the device is on, its source voltage is +5V. To maintain the on state, the gate has to be another 5 or 10V above that. The “12V” supply is not particularly critical. A minimally filtered, unregulated rectifier will suffice.

**BATTERY BACKUP SYSTEMS**

Here we consider circuits that normally draw power from the AC line, but switch to battery operation in the event of a power failure. We assume that in battery operation high-current loads will be allowed to die along with the AC power. The system may continue then with reduced functionality, monitoring a control transducer, perhaps, or driving an LCD. Or it may go into a bare-bones survival mode, in which critical data is saved but nothing else happens till AC power is restored.

In any case it is necessary to have some early warning of an impending power failure so that the system can arrange an orderly transfer to battery power. Early warning systems can operate by monitoring either the AC line voltage or the unregulated rectifier output, or even by monitoring the regulated DC voltage.

Monitoring the AC line voltage gives the earliest warning. That way you can know within one or two half-cycles of line frequency that AC power is down. In most cases you then have at least another half-cycle of line frequency before the regulated V_{CC} starts to fall. In a half-cycle of line frequency an 80C51BH can execute about 5,000 instructions—plenty of time to arrange an orderly transfer of power.

The circuit in Figure 11 uses a Zener diode to test the line voltage each half-cycle, and a junction transistor to pass the information on to the 80C51BH. (Obviously a voltage comparator with a suitable reference source can
perform the same function, if one prefers.) The way it works is if the line voltage reaches an acceptably high level, it breaks over Z1, drives Q1 to saturation, and interrupts the 80C51BH. The interrupt would be transition-activated, in this application. The interrupt service routine reloads one of the C51BH's timers to a value that will make it roll over in something between one and two half-cycles of line frequency. As long as the line voltage is healthy, the timer never rolls over, because it is reloaded every half-cycle. If there is a single half-cycle in which the line voltage doesn’t reach a high enough level to generate the interrupt, the timer rolls over and generates a timer interrupt.

The timer interrupt then commences the transition to battery backup. Critical data needs to be copied into protected RAM. Signals to circuits that are going to lose power must be written to logic low. Protected circuits (those powered by VCC2) that communicate with unprotected circuits must be deselected. The microcontroller itself may be put into Idle, so that it can continue some level of interrupt-driven functionality, or it may be put into Power Down.

Note that if the CPU is going to invoke Power Down, the Special Function Registers may also need to be copied into protected RAM, since the reset that terminates the Power Down mode will also initialize all the SFRs to their reset values.

The circuit in Figure 11 does not show a wake-up mechanism. A number of choices are available, however. A pushbutton could be used to generate an interrupt, if the CPU is in Idle, or to activate reset, if the CPU is in Power Down.

Automatic wake-up on power restoration is also possible. If the CPU is in Idle, it can continue to respond to any interrupts that might be generated by Q1. The interrupt service routine determines from the status of flag bits GF0 and GF1 in PCON that it is in Idle because there was a power outage. It can then sample VCC1 through a voltage comparator similar to Z1, Q1 in Figure 11. A satisfactory level of VCC1 would be indicated by the transistor being in saturation.

But perhaps you can’t spare the timer that is the key to the operation of the circuit in Figure 11. In that case a retriggerable one-shot, triggered by the AC line voltage, can perform essentially the same function. Figure 12 shows an example of this type of power failure detector. A retriggerable one-shot (one half of a 74HC123) monitors the AC line voltage through transistor Q1. Q1 re-triggers the one-shot every half-cycle of line frequency. If the output pulse width is between one and two half-cycles of line frequency, then a single missing or low half-cycle will generate an active low warning flag, which can be used to interrupt the microcontroller.

The interrupt routine takes care of the transition to battery backup. From this point VCC1 may or may not actually drop out. The missing half-cycle of line voltage that caused the power down sequence may have been nothing more than a short glitch. If the AC line comes back strong enough to trigger the one-shot while VCC1 is still up (as indicated by the state of transistor Q2), then the other half of the 74HC123 will generate a wake-up signal.
Having been awakened, the 80C51BH will stay awake for at least another half-cycle of line frequency (another 5,000 or so instructions) before possibly being told to arrange another transfer of power. Consequently, if the line voltage is jittering erratically around the switchover point (determined by diode Z1), the system will limp along executing in half-cycle units of line frequency.

On the other hand, if the power outage is real and lengthy, Vcc1 will eventually fall below the level at which the backup battery takes over. The backup battery maintains power to the 80C51BH, and to the 74HC123, and to whatever other circuits are being protected during this outage. The battery voltage must be high enough to maintain VccMIN specs to the 80C51BH.

If the microcontroller is an 80C31BH, executing out of external ROM, and if the C31BH is put into Idle during the power outage, then the external ROM must also be supplied by the battery. On the other hand, if the C31BH is put into Power Down during the outage, then the ROM can be allowed to die with the AC power. The considerations here are the same as in Figure 9: Vcc to the ROM is still up at the time Power Down is invoked, and we must ensure (through selection of diode Z2 in Figure 12) that the 80C31BH is not awakened till ROM power is back in spec.

**POWER SWITCHOVER CIRCUITS**

Battery backup systems need to have a way for the protected circuits to draw power from the line-operated power supply when that source is available, and to switch over to battery power when required. The switchover circuit is simple if the entire system is to be battery powered in the event of a line power outage. In that case a pair of diodes suffice, as shown in Figure 12, provided VccMIN specs are still met after the diode drop has been subtracted from its respective power source.

The situation becomes more complicated when part of the circuit is going to be allowed to die when the AC power goes out. In that case it is difficult to maintain equal Vccs to protected and unprotected circuits (and possibly dangerous not to).

The problem can be alleviated by using a Schottky diode instead of a 1N4001, for its lower forward voltage drop. The 1N5820, for example, has a forward drop of about 0.35V at 1A.

Other solutions are to use a transistor or power MOSFET switch, as shown in Figure 13. With minor modifications this switch can be controlled by port pins.
80C31BH + CHMOS EPROM

The 27C64 and 87C64 are Intel's 8K byte CHMOS EPROMs. The 27C64 requires an external address latch, and can be used with the 80C31BH as shown in Figure 14a. In most 8031 + 2764 (HMOS) applications, the 2764's Chip Enable (CE) pin is hardwired to ground (since it's normally the only program memory on the bus). This can be done with the CHMOS versions as well, but there is some advantage in connecting CE to ALE, as shown in Figure 14a. The advantage is that if the 80C31BH is put into Idle mode, since ALE goes to a 1 in that mode, the 27C64 will be deselected and go into a low current standby mode.

The timing waveforms for this configuration are shown in Figure 14b. In Figure 14b the signals and timing parameters in parenthesis are those of the 27C64, and the others are of the 80C31BH, except Tprop is a parameter of the address latch. The requirements for timing compatibility are

\[
\begin{align*}
T_{AVIV} - T_{prop} &> t_{ACC} \\
T_{LLIV} &> t_{CE} \\
T_{LIV} &> t_{OE} \\
T_{XIZ} &> t_{DF}
\end{align*}
\]

If the application is going to use the Power Down mode then we have another consideration: In Idle, ALE = PSEN = 1, and in Power Down, ALE = PSEN = 0. In a realistic application there are likely to be more chips in the circuit than are shown in Figure 14, and it is likely that the nonessential ones will have their VCC removed while the CPU is in Power Down. In that case the EPROM and the address latch should be among the chips that have VCC removed, and logic lows are exactly what are required at ALE and PSEN.

But if VCC is going to be maintained to the EPROM during Power Down, then it will be necessary to de-
select the EPROM when the CPU is in Power Down. If Idle is never invoked, CE of the EPROM can be connected to P2.7 of the 80C31BH, as shown in Figure 15a. In normal operation, P2.7 will be emitting the MSB of the Program Counter, which is 0 if the program contains less than 32K of code. Then when the CPU goes into Power Down, the Port 2 pins emit P2 SFR data, which puts a 1 at P2.7, thus deselecting the EPROM.

If Idle and Power Down are both going to be used, CE of the EPROM can be driven by the logical OR of ALE and P2.7, as shown in Figure 15b. In Idle, ALE = 1 will deselect the EPROM, and in Power Down, P2.7 = 1 will deselect it.

The 87C64 is like the 27C64 except that it has an on-chip address latch. The Port 0 pins are tied to both address and data pins of the 87C64, as shown in Figure 16a. ALE drives the EPROM's ALE/CS input. During ALE high, the address information is allowed to flow into the EPROM and begin accessing the code byte. On the falling edge of ALE the address byte is internally latched. The A0–A7 inputs are then ignored and the same bus lines are used to transmit the fetched code byte from the 00-07 pins back to the 80C31BH.

The timing waveforms for this configuration are shown in Figure 16b. In Figure 16b the signals and timing parameters in parentheses are those of the 87C64, and the others are of the 80C31BH. The requirements for timing compatibility are

- \( T_{LHLL} > t_{LL} \)
- \( T_{AVLL} > t_{AL} \)
- \( T_{LLAX} > t_{LA} \)
- \( T_{LLIV} > t_{ACL} \)
- \( T_{PLIV} > t_{OE} \)
- \( T_{LLPL} > t_{COE} \)
- \( T_{PKIZ} > t_{OHZ} \)

The same considerations apply to the 87C64 as to the 27C64 with regards to the Idle and Power Down modes. Basically you want \( CS = 1 \) if \( V_{CC} \) is maintained to the EPROM, and \( CS = OE = 0 \) if \( V_{CC} \) is removed.

**SCANNING A KEYBOARD**

There are many different kinds of keyboards, but alphanumeric keyboards generally consist of a matrix of 8 scan lines and 8 receive lines as shown in Figure 17. Each set of lines connects to one port of the microcontroller. The software has written 0s to the scan lines, and 1s to the receive lines. Pressing a key connects a scan line to a receive line, thus pulling the receive line to a logic low.

The 8 receive lines are ANDed to one of the external interrupt pins, so that pulling any of the receive lines low generates an interrupt. The interrupt service routine has to identify the pressed key, if only one key is down, and convert that information to some useful output. If more than one key in the line matrix is found to be pressed, no action is taken. (This is a "two key lockout" scheme.)
On some keyboards, certain keys (Shift, Control, Escape, etc.) are not a part of the line matrix. These keys would connect directly to a port pin on the microcontroller, and would not cause lock-out if pressed simultaneously with a matrix key, nor generate an interrupt if pressed singly.

Normally the microcontroller would be in idle mode when a key has not been pressed, and another task is not in progress. Pressing a matrix key generates an interrupt, which terminates the Idle. The interrupt service routine would first call a 30 ms (or so) delay to debounce the key, and then set about the task of identifying which key is down.

First, the current state of the receive lines is latched into an internal register. If a single key is down, all but one of the lines would be read as 1s. Then 0s are written to the receive lines and 1s to the scan lines, and the scan lines are read. If a single key is down, all but one of...
these lines would be read as 1s. By locating the single 0 in each set of lines, the pressed key can be identified. If more than one matrix key is down, one or both sets of lines will contain multiple 0s.

A subroutine is used to determine which of 8 bits in either set of lines is 0, and whether more than one bit is 0. Figure 18 shows a subroutine (SCAN) which does that using the 8051's bit-addressing capability. To use the subroutine, move the line data into a bit-addressable RAM location named LINE, and call the SCAN routine. The number of LINE bits which are zero is returned in ZERO_COUNTER. If only one bit is zero, its number (1 through 8) is returned in ZERO_BIT.

The interrupt service routine that is executed in response to a key closure might then be as follows:

```
RESPONSE_TO_KEY_CLOSURE:
    CALL DEBOUNCE_DELAY
    MOV LINE, P1; ;See Figure 17.
    CALL SCAN
    DJNZ ZERO_COUNTER, REJECT
    MOV ADDRESS, ZERO_BIT
    MOV P2, #0FFH; ;See Figure 17.
    MOV P1, #0
    MOV LINE, P2
    CALL SCAN
    DJNZ ZERO_COUNTER, REJECT
    XCH A, ZERO_BIT
    SWAP A
    ORL ADDRESS, A
    XCH A, ZERO_BIT
    MOV P1, #0FFH
    MOV P2, #0
    REJECT: CLR EX0
    RETI
```

![Figure 17. Scanning a Keyboard](image-url)
Notice that RESPONSE_TO_KEY_CLOSURE does not change the Accumulator, the PSW, nor any of the registers R0 through R7. Neither do SCAN or DEBOUNCE_DELAY.

What we come out with then is a one-byte key address (ADDRESS) which identifies the pressed key. The key's scan line number is in the upper nibble of ADDRESS, and its receive line number is in the lower nibble. ADDRESS can be used in a look-up table to generate a key code to transmit to a host computer, and/or to a display device.

The keyboard interrupt itself must be edge-triggered, rather than level-activated, so that the interrupt routine is invoked when a key is pressed, and is not constantly being repeated as long as the key is held down. In edge-triggered mode, the on-chip hardware clears the interrupt flag (EX0, in this case) as the service routine is being vectored to. In this application, however, contact bounce will cause several more edges to occur after the service routine has been vectored to, during the DEBOUNCE_DELAY routine. Consequently it is necessary to clear EX0 again in software before executing RETI.

The debounce delay routine also takes advantage of the Idle mode. In this routine a timer must be preloaded with a value appropriate to the desired length of delay. This would be

\[
\text{timer preload} = \frac{(\text{osc kHz}) \times (\text{delay time ms})}{12}
\]

For example, with a 3.58 MHz oscillator frequency, a 30 ms delay could be obtained using a preload value of \(-8950\), or DDOA, in hex digits.

In the debounce delay routine (Figure 19), the timer interrupt is enabled and set to a higher priority than the keyboard interrupt, because as we invoke Idle, the keyboard interrupt is still "in progress". An interrupt of the same priority will not be acknowledged, and will not terminate the Idle mode. With the timer interrupt set to priority 1, while the keyboard interrupt is a priority 0, the timer interrupt, when it occurs, will be acknowledged and will wake up the CPU. The timer interrupt service routine does not itself have to do anything. The service routine might be nothing more than a single RETI instruction. RETI from the timer interrupt service routine then returns execution to the debounce delay routine, which shuts down the timer and returns execution to the keyboard service routine.

**DRIVING AN LCD**

An LCD (Liquid Crystal Display) consists of a backplane and any number of segments or dots which will be used to form the image being displayed. Applying a voltage (nominally 4 or 5V) between any segment and the backplane causes the segment to darken. The only catch is that the polarity of the applied voltage has to be periodically reversed, or else a chemical reac-
DEBOUNCE_DELAY:

MOV T1, #TL1_PRELOAD ; Preload low byte.
MOV TH1, #TH1_PRELOAD ; Preload high byte.
SETB ET1 ; Enable Timer 1 interrupt.
SETB PT1 ; Set Timer 1 interrupt to high priority.
SETB TR1 ; Start timer running.
ORL PCON, #1 ; Invoke Idle mode.

The next instruction will not be executed until the delay times out.

CLR TRI ; Stop the timer.
CLR PTI ; Back to priority 0 (if desired).
CLR ETI ; Disable Timer 1 interrupt (if desired).
RETI

Figure 19. Subroutine DEBOUNCE_DELAY Puts the 80C51BH into Idle During the Delay Time

tion takes place in the LCD which causes deterioration and eventual failure of the liquid crystal.

To prevent this happening, the backplane and all the segments are driven with an AC signal, which is derived from a rectangular voltage waveform. If a segment is to be "off" it is driven by the same waveform as the backplane. Thus it is always at backplane potential. If the segment is to be "on" it is driven with a waveform that is the inverse of the backplane waveform. Thus it has about 5V of periodically changing polarity between it and the backplane.

With a little software overhead, the 80C51BH can perform this task without the need for additional LCD drivers. The only drawback is that each LCD segment uses up one port pin, and the backplane uses one more. If more than, say, two 7-segment digits are being driven, there aren't many port pins left for other tasks. Nevertheless, assuming a given application leaves enough port pins available to support this task, the considerations for driving the LCD are as follows.

Suppose, for example, it is a 2-digit display with a decimal point. One port (TENS_DIGIT) connects to the 7 segments of the tens digit plus the backplane. Another port (ONES_DIGIT) connects to a decimal point plus the 7 segments of the ones digit.

One of the 80C51BH's timers is used to mark off half-periods of the drive voltage waveform. The LCD drive waveform should have a rep rate between 30 and 100 Hz, but it's not very critical. A half-period of 12 ms will set the rep rate to about 42 Hz. The preload/reload value to get 12 ms to rollover is the 2's complement negative of the oscillator frequency in kHz: if the oscillator frequency is 3.58 MHz, the reload value is -3580, or F204 in hex digits.

Now, the 80C51BH would normally be in Idle, to conserve power, during the time that the LCD and other tasks are not requiring servicing. When the timer rolls over it generates an interrupt, which brings the 80C51BH out of Idle. The service routine reloads the timer (for the next rollover), and inverts the logic levels of all the pins that are connected to the LCD. It might look like this:

LCD_DRIVE_INTERRUPT:

MOV TL1, #LOW(- XTAL_FREQ)
MOV TH1, #HIGH(- XTAL_FREQ)
XRL TENS_DIGIT, #0FFH
XRL ONES_DIGIT, #0FFH
RETI

To update the display, one would use a look-up table to generate the characters. In the table, "on" segments are represented as 1s, and "off" segments as 0s. The backplane bit is represented as a 0. The quantity to be displayed is stored in RAM as a BCD value. The look-up table operates on the low nibble of the BCD value, and produces the bit pattern that is to be written to either the ones digit or the tens digit. Before the new patterns can be written to the LCD, the LCD drive interrupt has to be disabled. That is to prevent a polarity reversal from taking place between the times the two digits are written. An update subroutine is shown in Figure 20.

USING AN LCD DRIVER

As was noted, driving an LCD directly with an 80C51BH uses a lot of port pins. LCD drivers are available in CMOS to interface an 80C51BH to a 4-digit display using only 7 of the C51BH's I/O pins. Basically, the C51BH tells the LCD driver what digit is to be displayed (4 bits) and what position it is to be displayed in (2 bits), and toggles a Chip Select pin to tell the driver to latch this information. The LCD driver generates the display characters (hex digits), and takes care of the polarity reversals using its own RC oscillator to generate the timing.
Figure 25 shows an 80C51BH working with an ICM7211M to drive a 4-digit LCD, and the software that updates the display.

One could equally well send information to the LCD driver over the bus. In that case, one would set up the Accumulator with the digit select and data input bits, and execute a MOVX® R0,A instruction. The LCD driver's chip select would be driven by the CPU's WR signal. This is a little easier in software than the direct bit manipulation shown in Figure 21. However, it uses more I/O pins, unless there is already some external memory involved. In that case, no extra pins are used up by adding the LCD driver to the bus.

RESONANT TRANSDUCERS

Analog transducers are often used to convert the value of a physical property, such as temperature, pressure, etc., to an analog voltage. These kinds of transducers then require an analog-to-digital converter to put the measurement into a form that is compatible with a digital control system. Another kind of transducer is now becoming available that encodes the value of the physical property into a signal that can be directly read by a digital control system. These devices are called resonant transducers.

Resonant transducers are oscillators whose frequency depends in a known way on the physical property being measured. These devices output a train of rectangular pulses whose repetition rate encodes the value of the quantity being measured. The pulses can in most cases be fed directly into the 80C51BH, which then measures either the frequency or period of the incoming signal, basing the measurement on the accuracy of its own clock oscillator. The 80C51BH can even do this in its sleep; that is, in Idle.

When the frequency or period measurement is completed, the C51BH wakes itself up for a very short time to perform a sanity check on the measurement and convert it in software to any scaling of the measured quantity that may be desired. The software conversion can include corrections for nonlinearities in the transducer's transfer function.

Resolution is also controlled by software, and can even be dynamically varied to meet changing needs as a situation becomes more critical. For example, in a process controller you can increase your resolution ("fine tune" the control, as it were) as the process approaches its target.

The nominal reference frequency of the output signal from these devices is in the range of 20 Hz to 500 kHz, depending on the design. Transducers are available that have a full scale frequency shift 2 to 1. The transducer operates from a supply voltage range of 3V to 20V, which means it can operate from the same supply voltage as the 80C51BH. At 5V, the transducer draws less than 5 mA (Reference 7). It can normally be connected directly to one of the C51BH's port pins, as shown in Figure 22.

FREQUENCY MEASUREMENTS

Measuring a frequency means counting pulses for a known sample time. Two timer/counters can be used, one to mark off the sample time and one to count pulses. If the frequency being counted doesn’t exceed 50 kHz or so, one may equally well connect the transducer signal to one of the external interrupt pins, and count pulses in software. That frees up one timer, with very little cost in CPU time.

The count that is directly obtained is TxF, where T is the sample time and F is the frequency. The full scale

```
UPDATE_LCD
CLR ET1
MOV DPTR,#TABLE_ADDRESS
MOV A,BCD_VALUE
SHAP A
ANL A,#0FH
MDVC A,#A+DPTR
MDV TENS_DIGIT,A
MOV A,BCD_VALUE
ANL A,#0FH
MDVC A,#A+DPTR
MDV C.DECIMAL_POINT
MDV ACC,7,C
MDV ONES_DIGIT,A
SETB ET1
RET
```

Figure 20. UPDATE_LCD Routine Writes Two Digits to an LCD
range is $T_x(F_{\text{max}} - F_{\text{min}})$. For n-bit resolution

$$1 \text{ LSB} = \frac{T_x(F_{\text{max}} - F_{\text{min}})}{2^n}$$

Therefore the sample time required for n-bit resolution is

$$T = \frac{2^n}{F_{\text{max}} - F_{\text{min}}}$$

For example, 8-bit resolution in the measurement of a frequency that varies between 7 kHz and 9 kHz would require, according to this formula, a sample time of 128 ms. The maximum acceptable frequency count would be $128 \text{ ms} \times 9 \text{ kHz} = 1152 \text{ counts}$. The minimum would be 896 counts. Subtracting 896 from each frequency count (or presetting the frequency counter to $-896 = 0\text{FC80H}$) would allow the frequency to be reported on a scale of 0 to FF in hex digits.

---

**Figure 21a. Using an LCD Driver**

**UPDATE_LCD:**

```
MOV A, DISPLAY_HI   ; High byte of 4-digit display.
SETB DIGIT_SELECT_2 ; Select leftmost digit of LCD.
SETB DIGIT_SELECT_1 ; (Digit address = 11B.)
CALL SHIFT_AND_LOAD ; High nibble of high byte to selected digit.
CALL SHIFT_AND_LOAD ; Low nibble of high byte to selected digit.
MOV A, DISPLAY_LO   ; Low byte of 4-digit display.
SETB DIGIT_SELECT_2 ; Select third digit of LCD.
SETB DIGIT_SELECT_1 ; (Digit address = 01B.)
CALL SHIFT_AND_LOAD ; High nibble of low byte to selected digit.
CALL SHIFT_AND_LOAD ; Low nibble of low byte to selected digit.
RET
```

**SHIFT_AND_LOAD:**

```
RLC A               ; MSB to carry bit (CY)
MOV DATA_INPUT_B3.C ; CY to Data Input pin B3.
RLC A               ; Next bit to CY.
RLC A               ; Next bit to CY.
RLC A               ; Last bit to CY.
MOV DATA_INPUT_B0.C ; CY to Data Input pin B0.
CLR CHIP_SELECT     ; Toggle Chip Select.
SETB CHIP_SELECT    ; 0-to-1 transition latches info.
RET
```

**Figure 21b. UPDATE_LCD Routine Writes 4 Digits to an LCD Driver**
To implement the measurement, one timer is used to establish the sample time. The timer is preset to a value that causes it to roll over at the end of the sample time, generating an interrupt and waking the CPU from its Idle mode. The required preset value is the 2's complement negative of the sample time measured in machine cycles. The conversion from sample time to machine cycles is to multiply it by 1/12 the clock frequency. For example, if the clock frequency is 12 MHz, then a sample time of 128 ms is

\[(128 \text{ ms}) \times (12000 \text{ kHz})/12 = 128000 \text{ machine cycles.}\]

Then the required preset value to cause the timer to roll over in 128 ms is

\[-128000 = \text{FE0C00}, \text{ in hex digits.}\]

Note that the preset value is 3 bytes wide whereas the timer is only 2 bytes wide. This means the timer must be augmented in software in the timer interrupt routine to three bytes. The 80C51BH has a DJNZ instruction (decrement and jump if not zero) that makes it easier to code the third timer byte to count down instead of up. If the third timer byte counts down, its reload value is the 2's complement of what it would be for an up-counter. For example, if the 2's complement of the sample time is FE0C00, then the reload value for the third timer byte would be 02, instead of FE. The timer interrupt routine might then be:

```
TIMER_INTERRUPT_ROUTINE:
    DJNZ THIRD_TIMER_BYTE, OUT
    MOV TLO, #O
    MOV THO, #0CH
    MOV THIRD_TIMER_BYTE, #2
    MOV FREQUENCY, COUNTER_LO

;Preset COUNTER to -896:
    MOV COUNTER_LO, #80H
    MOV COUNTER_HI, #0FH

OUT: RETI
```

At this point the value of the frequency of the transducer signal, measured to 8 bit resolution, is contained in FREQUENCY. Note that the timer can be reloaded on the fly. Note too that the timer can be reloaded on the fly. Note too that for 8-bit resolution only the low byte of the frequency counter needs to be read, since the high byte is necessarily 0. However, one may want to test the high byte to ensure that it is zero, as a sanity check on the data. Both bytes, of course must be reloaded.

### PERIOD MEASUREMENTS

Measuring the period of the transducer signal means measuring the total elapsed time over a known number, \(N_s\), of transducer pulses. The quantity that is directly measured is \(N_T\), where \(T\) is the period of the transducer signal in machine cycles. The relationship between \(T\) in machine cycles and the transducer frequency \(F\) in arbitrary frequency units is

\[T = \frac{F_{\text{xtal}}}{F} \times \frac{1}{12},\]

where \(F_{\text{xtal}}\) is the 80C51BH clock frequency, in the same units as \(F\).

The full scale range then is \(N_s (T_{\text{max}}- T_{\text{min}})\). For \(n\)-bit resolution,

\[1 \text{ LSB} = \frac{N_s (T_{\text{max}}- T_{\text{min}})}{2^n}\]

Therefore the number of periods over which the elapsed time should be measured is

\[N = \frac{2^n}{T_{\text{max}}- T_{\text{min}}}\]

However, \(N\) must also be an integer. It is logical to evaluate the above formula (don't forget \(T_{\text{max}}\) and \(T_{\text{min}}\) have to be in machine cycles) and select for \(N\) the next higher integer. This selection gives a period measurement that has somewhat more than \(n\)-bit resolution, but it can be scaled back if desired.

For example, suppose we want 8-bit resolution in the measurement of the period of a signal whose frequency varies from 7.1 kHz to 9 kHz. If the clock frequency is 12 MHz, then \(T_{\text{max}}\) is \((12000 \text{ kHz}/7.1 kHz) \times (1/12) = 141\) machine cycles. \(T_{\text{min}}\) is 111 machine cycles. The required value for \(N\), then, is \(256/(141-111) = 8.53\) periods, according to the formula. Using \(N = 9\) periods will give a maximum \(N T\) value of \(141 \times 9 = 1269\) machine cycles. The minimum \(N T\) will be \(111 \times 9 = 999\) machine cycles. A lookup table can be used to
scale these values back to a range of 0 to 255, giving precisely the 8-bit resolution desired.

To implement the measurement, one timer is used to measure the elapsed time, NT. The transducer is connected to one of the external interrupt pins, and this interrupt is configured to the transition-activated mode. In the transition-activated mode every 1-to-0 transition in the transducer output will generate an interrupt. The interrupt routine counts transducer pulses, and when it gets to the predetermined N, it reads and clears the timer. For the specific example cited above, the interrupt routine might be:

```
INTERRUPT_RESPONSE:
DJNZ N, OUT
MOV N, #9
CLR EA
CLR TRI
MOV NT_LO, TLI
MOV NT_HI, THI
MOV TLI, #9
MOV THI, #0
SETB TRI
SETB EA
CALL LOOKUP_TABLE
OUT: RETI
```

In this routine a pulse counter N is decremented from its preset value, 9, to zero. When the counter gets to zero it is reloaded to 9. Then all interrupts are blocked for a short time while the timer is read and cleared. The timer is stopped during the read and clear operations, so “clearing” it actually means presetting it to 9, to make up for the 9 machine cycles that are missed while the timer is stopped.

The subroutine `LOOKUP_TABLE` is used to scale the measurement back to the desired 8-bit resolution. It can also include built-in corrections for errors or nonlinearities in the transducer’s transfer function.

The subroutine uses the `MOVC A, @A + DPTR` instruction to access the table, which contains 270 entries commencing at the 16-bit address referred to as TABLE. The subroutine must compute the address of the table entry that corresponds to the measured value of NT. This address is

```
DPTR = TABL + NT - NTMIN,
```

where NTMIN = 999, in this specific example.

```
LOOKUP_TABLE:
PUSH ACC
PUSH PSW
MOV A, #LOW(TABLE-NTMIN)
ADD A, NT_LO
MOV DPL, A
MOV A, #HIGH(TABLE-NTMIN)
ADD A, NT_HI
POP PSW
POP ACC
```

At this point the value of the period of the transducer signal, measured to 8 bit resolution, is contained in PERIOD.

**PULSE WIDTH MEASUREMENTS**

The 80C51BH timers have an operating mode which is particularly suited to pulse width measurements, and will be useful in these applications if the transducer signal has a fixed duty cycle.

In this mode the timer is turned on by the on-chip circuitry in response to an input high at the external interrupt pin, and off by an input low, and it can do this while the 80C51BH is in Idle. (The “GATE” mode of timer operation is described in the Intel Microcontroller Handbook.) The external interrupt itself can be enabled, so the same I-to-O transition from the transducer that turns off the timer also generates an interrupt. The interrupt routine then reads and resets the timer.

The advantage of this method is that the transducer signal has direct access to the timer gate, with the result that variations in interrupt response time have no effect on the measurement.

Resonant transducers that are designed to fully exploit the GATE mode have an internal divide-by-N circuit that fixes the duty cycle at 50% and lowers the output frequency to the range of 250 to 500 Hz (to control RFI). The transfer function between transducer period and measurand is approximately linear, with known and repeatable error functions.

**HMOS/CHMOS Interchangeability**

The CHMOS version of the 8051 is architecturally identical with the HMOS version, but there are nevertheless some important differences between them which the designer should be aware of. In addition, some applications require interchangeability between HMOS and CHMOS parts. The differences that need to be considered are as follows:

**External Clock Drive:** To drive the HMOS 8051 with an external clock signal, one normally grounds the XTAL1 pin and drives the XTAL2 pin. To drive the CHMOS 8051 with an external clock signal, one must drive the XTAL1 pin and leave the XTAL2 pin unconnected. The reason for the difference is that in the
HMOS 8051, it is the XTAL2 pin that drives the internal clocking circuits, whereas in the CHMOS version it is the XTAL1 pin that drives the internal clocking circuits.

There are several ways to design an external clock drive to work with both types. For low clock frequencies (below 6 MHz), the HMOS 8051 can be driven in the same way as the CHMOS version, namely, through XTAL1 with XTAL2 unconnected. Another way is to drive both XTAL1 and XTAL2; that is, drive XTAL1 and use and external inverter to derive from XTAL1 a signal with which to drive XTAL2.

In either case, a 74HC or 74HCT circuit makes an excellent driver for XTAL1 and/or XTAL2, because neither the HMOS nor the CHMOS XTAL pins have TTL-like input logic levels.

Unused Pins: Unused pins of Ports 1, 2 and 3 can be ignored in both HMOS and CHMOS designs. The internal pullups will put them into a defined state. Unused Port 0 pins in 8051 applications can be ignored, even if they’re floating. But in 80C51BH applications, these pins should not be left afloat. They can be externally pulled up or down, or they can be internally pulled down by writing Os to them.

8031/80C31BH designs may or may not need pullups on Port 0. Pullups aren’t needed for program fetches, because in bus operations the pins are actively pulled high or low by either the 8031 or the external program memory. But they are needed for the CHMOS part if the Idle or Power Down mode is invoked, because in these modes Port 0 floats.

Logic Levels: If VCC is between 4.5 V and 5.5 V, an input signal that meets the HMOS 8051’s input logic levels will also meet the CHMOS 80C51BH’s input logic levels (except for XTAL1/XTAL2 and RST). For the same VCC condition, the CHMOS device will reach or surpass the output logic levels of the HMOS device. The HMOS device will not necessarily reach the output logic levels of the CHMOS device. This is an important consideration if HMOS/CHMOS interchangeability must be maintained in an otherwise CMOS system.

HMOS 8051 outputs that have internal pullups (Ports 1, 2, and 3) “typically” reach 4 V or more if IOH is zero, but not fast enough to meet timing specs. Adding an external pullup resistor will ensure the logic level, but still not the timing, as shown in Figure 23. If timing is an issue, the best way to interface HMOS to CMOS is through a 74HCT circuit.

Idle and Power Down: The Idle and Power Down modes exist only on the CHMOS devices, but if one wishes to preserve the capability of interchanging HMOS and CHMOS 8051s, the software has to be designed so that the HMOS parts will respond in an acceptable manner when a CHMOS reduced power mode is invoked.

For example, an instruction that invokes Power Down can be followed by a “JMP $”:

```
CLR EA
ORL PCON,#2
JMP $
```

The CHMOS and HMOS parts will respond to this sequence of code differently. The CHMOS part, going into a normal CHMOS Power Down Mode, will stop fetching instructions until it gets a hardware reset. The HMOS part will go through the motions of executing the ORL instruction, and then fetch the JMP instruction. It will continue fetching and executing JMP $ until hardware reset.

Maintaining HMOS/CHMOS 8051 interchangeability in response to Idle requires more planning. The HMOS part will not respond to the instruction that puts the CHMOS part into Idle, so that instruction needs to be followed by a software idle. This would be an idling loop which would be terminated by the same conditions that would terminate the CHMOS’s hardware Idle. Then when the CHMOS device goes into Idle, the HMOS version executes the idling loop, until either a hardware reset or an enabled interrupt is received. Now if Idle is terminated by an interrupt, execution for the CHMOS device will proceed after RETI from the instruction following the one that invoked Idle. The instruction following the one that invoked Idle is the idling loop that was inserted for the HMOS device. At this point, both the HMOS and CHMOS devices must be able to fall through the loop to continue execution.
One way to achieve the desired effect is to define a "fake" Idle flag, and set it just before going into Idle. The instruction that invoked Idle is followed by a software idle:

```
SETB IDLE
ORL PCON,#1
JB IDLE,$
```

Now the interrupt that terminates the CHMOS’s Idle must also break the software idle. It does so by clearing the “Idle” bit:

```
...  IDLE
CLR IDLE
RETI
```

Note too that the PCON register in the HMOS 8051 contains only one bit, SMOD, whereas the PCON register in CHMOS contains SMOD plus four other bits. Two of those other bits are general purpose flags. Maintaining HMOS/CHMOS interchangeability requires that these flags not be used.

REFERENCES
Enhanced Serial Port
on the 83C51FA

BETSY JONES
ECO APPLICATIONS ENGINEER
The serial port on the 8051 has been enhanced on the 83C51FA with the addition of two new features: Automatic Address Recognition and Framing Error Detection. Automatic Address Recognition facilitates multiprocessor communications by reducing CPU overhead. Framing Error Detection increases communication reliability by checking each reception for a valid stop bit.

This Application Note explains how to use these new features with samples of code for typical applications. A section is also included which reviews how to set up the serial port for multiprocessor applications.

MULTIPROCESSOR COMMUNICATIONS

In applications where multiple controllers jointly perform a task, the master controller must be able to communicate selectively with individual slaves. To do this, the master first identifies the target slave (or slaves) with an address byte and then transmits a block of data. The target slaves must be able to identify their own address before receiving any data bytes.

The serial port on the 8051 provides a 9-bit mode to facilitate multiprocessor communication. The 9th bit allows the controller to distinguish between address and data bytes. In this mode, a total of 11 bits are received or transmitted: a start bit (0), 8 data bits (LSB first), a programmable 9th bit, and a stop bit (1). See Figure below.

The 9th bit is set to 1 to identify address bytes and set to 0 for data bytes. A typical data stream is seen below:

ADDRESS BYTE / DATA BYTE / DATA BYTE / . . .
D8 = 1  D8 = 0  D8 = 0

Initially the slave is set up to only receive address bytes. Once it receives its own address, the slave reconfigures itself to receive data. On the 8051 serial port, an address byte interrupts all slaves for an address comparison. On the 83C51FA, however, Automatic Address Recognition allows the addressed slave to be the only one interrupted; that is, the address comparison occurs in hardware, not software. With this feature, the master controller can establish communication with one or more slaves without all the slaves having to respond to the transmission.

AUTOMATIC ADDRESS RECOGNITION

Automatic Address Recognition reduces the CPU time required to service the serial port. Since the CPU is only interrupted when it receives its own address, the software overhead to compare addresses is eliminated. This would also effectively reduce the sophistication of the serial protocol when numerous controllers are sharing the same serial link.

This same feature can also be used in conjunction with the Idle Mode to reduce the system's overall power consumption. For instance, a master may need to communicate with only one slave at a time. With all slaves in Idle Mode, only that one slave would be interrupted to respond to the master's transmission. Without Automatic Addressing, each slave would have to "wake up" to check for its address. Limiting the interruptions reduces the amount of current drawn by the system and thus reduces the power consumption.

In multiprocessor applications the serial port is configured in either of the 9-bit modes (Mode 2 or 3). Mode 2 has a fixed baud rate whereas Mode 3 is variable. For more information on the different serial port modes refer to the "Serial Port Set Up" section.

Automatic Address Recognition is enabled by setting the SM2 bit in SCON. Each slave has its SM2 bit set waiting for an address byte (9th bit = 1). The Receive Interrupt (RI) flag will get set when the received byte corresponds to either a Given or Broadcast Address. The slave then clears its SM2 bit to enable reception of data bytes (9th bit = 0) from the master.

The master can selectively communicate with groups of slaves by using the Given Address. Addressing all slaves at once is possible with the Broadcast Address. These addresses are defined for each slave by two new Special Function Registers: SADDR and SADEN.

A slave's individual address is specified in SADDR. SADEN is a mask byte that defines don't-cares to form the Given Address. These don't-cares allow flexibility in the user-defined protocol to address one or more slaves. The following is an example of how to define Given Addresses and selectively address different slaves.

![Serial Port Diagram](270490-1)
FRAMING ERROR DETECTION

Framing Error Detection is another new feature on 83C51FA serial port which allows the receiving controller to check for valid stop bits in Modes 1, 2, or 3. A missing stop bit can be caused, for example, by noise on the serial lines or transmission by two CPUs simultaneously.

If a stop bit is missing a Framing Error bit FE will be set. This bit can then be checked in software after each reception to detect communication errors. Once set, the FE bit must be cleared in software. A valid stop bit will not clear FE.

The FE bit is located in SCON and shares the same bit address as SM0. To determine which is accessed, a new control bit called SMOD0 has been added in the PCON register (see figures below). If SMOD0 = 0, then accesses to SCON.7 are to SM0. If SMOD0 = 1, then accesses to SCON.7 are to FE.

PCON: Power Control Register (Not Bit Addressable)

<table>
<thead>
<tr>
<th>SMOD1</th>
<th>SMOD0</th>
<th>POF</th>
<th>GF1</th>
<th>GF0</th>
<th>PD</th>
<th>IDL</th>
</tr>
</thead>
</table>
Address = 87H

SCON: Serial Port Control Register (Bit Addressable)

| SM0/FE | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |

Address = 98H

SERIAL PORT SOFTWARE

The following sections of code show examples of how to invoke Automatic Addressing and Framing Error Detection. Routines for both the slave and master are given. Code is also included to initialize both serial ports; however, for more information on setting up the serial port refer to the next section.

For this example, the master and slave are transmitting/receiving at 9600 baud with a 12 MHz crystal frequency. To obtain this baud rate, the serial port is configured in Mode 3 and Timer 2 is used as the baud-rate generator.

Listing 1 shows the initialization for the slave. Notice that Automatic Addressing and Framing Error Detection are enabled. The Given and Broadcast addresses for this slave are taken from Slave 1 in the previous example. A temporary byte has also been defined to store the incoming data byte.

The slave will remain in Idle Mode until it is interrupted by its own address. At that point, it clears the SM2

---

Slave 1

<table>
<thead>
<tr>
<th>SADDR</th>
<th>SADEN</th>
<th>GIVEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0001</td>
<td>1111 1010</td>
<td>1111 0X0X</td>
</tr>
</tbody>
</table>

The SADEN bytes have been selected such that bit 1 (LSB) is a don’t-care for Slave 1’s Given Address, but bit 1 = 1 for Slave 2. Thus, to selectively communicate with just Slave 1 an address with bit 1 = 0 would be used (e.g. 1111 0000).

Similarly, bit 2 = 0 for Slave 1, but is a don’t-care for Slave 2. Now to communicate with just Slave 2 an address with bit 2 = 1 would be used (e.g. 1111 0111).

Finally, to communicate with both slaves at once the address must have bit 1 = 1 and bit 2 = 0. Notice, however, that bit 3 is a “don’t-care” for both slaves. This allows two different addresses to select both slaves (1111 0001 or 1111 0101). If a third slave was added that required its bit 3 = 0, then the latter address could be used to communicate with Slave 1 and 2 but not Slave 3.

The master can also communicate with all slaves at once with the Broadcast Address. It is formed from the logical OR of SADDR and SADEN with zeros defined as don’t-cares. For example, the Broadcast address for Slave 1 would be formed as follows:

<table>
<thead>
<tr>
<th>SADDR</th>
<th>SADEN</th>
<th>BROADCAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 0001</td>
<td>1111 1010</td>
<td>1111 1X11</td>
</tr>
</tbody>
</table>

The don’t-cares also allow flexibility in defining the Broadcast Address, but in most applications a Broadcast Address will be 0FFH.

SADDR and SADEN are located at address A9H and B9H, respectively. On Reset, SADDR and SADEN are initialized to 00H which defines the Given and Broadcast Addresses as XXXX XXXX (all don’t-cares). This assures the 83C51FA serial port to be backwards compatible with the other MCS®-51 products which do not implement Automatic Addressing.
Listing 1. Initialization Routine for the Slave

ORG 0OH
LJMP INIT

ORG 0023H
LJMP SERIAL_PORT_INTERRUPT

TEMP DATA 30H ; Temporary storage byte

INIT: MOV SCON, #0FOH ; Mode 3, enable Auto Addressing and reception
       ORL PCON, #40H ; FE bit accessed (SM2DO = 1)
       MOV RCAP2H, #OFFH ; Reload values for 9600 Baud
       MOV RCAP2L, #OD9H
       MOV T2CON, #34H ; Timer 2 set up, TR2 = 1 turns timer on

INTERRUPTS: SETB EA ; Enable global interrupt
             SETB ES ; Enable serial port interrupt

ADDRESSES: MOV SADDR, #11110001 ; Define Given & Broadcast Addresses
           MOV SADEN, #11111010
           ORL PCON, #01H

IDLE_MODE: ORL PCON, #01H ; Invoke Idle Mode

Listing 2. Receive Routine for the Slave

SERIAL_PORT_INTERRUPT:
  PUSH PSW
  CLR RI ; RI set when address is recognized & must be cleared in software
  CLR SM2 ; Reconfigure slave to receive data bytes

RECEIVE_DATA:
  JNB RI, $ ; Wait for RI to be set
  MOV C, SCON.7 ; Check for framing error
  JC FRAMING_ERROR
  MOV TEMP, SBUF ; Receive data byte & store in temporary location
  CLR RI ; Clear flag for next reception
  SETB SM2 ; Re-enable Automatic Addressing
  POP PSW
  RETI

FRAMING_ERROR:
  CLR SCON.7 ; Clear FE bit
  CLR C
  ; Error routine left up to the user
  POP PSW
  RETI
Listing 3. Initialization and Transmit Routines for the Master

| GIVEN_l | equ | 11110001B |
| MESSAGE_l | data | 30H |

**INIT:**
- MOV SCON, #ODOH  
  ; Mode 3, REN = 1
- MOV RCAP2H, #OFFH  
  ; 9600 Baud
- MOV RCAP2L, #OD9H
- MOV T2CON, #34H  
  ; Timer 2 set up, TR2 = 1

**TRANSMIT_ADDRESS:**
- CLR TI
- SETB TBB  
  ; Mark 1st byte as an address
- MOV SBUF, #GIVEN_l
- JNB TI, $  
  ; Send address
  ; Wait for transmission
  ; complete
- CLR TI
  ; Clear flag for next
  ; transmission

**TRANSMIT_DATA:**
- CLR TBB  
  ; Mark 2nd byte as a data
- MOV SBUF, MESSAGE_l
- JNB TI, $  
  ; Send data byte
- CLR TI

The initialization routine for the master in Listing 3 is very similar to the slave. In this example, however, the master does not need Automatic Addressing; it is simply transmitting address and data bytes. GIVEN_l is a byte to address the slave in the above example. MESSAGE_l is a register that contains the data byte sent to this slave. Its value is arbitrary for the sample code.

MODES 2 AND 3

Mode 2 which has a fixed baud rate and Mode 3 which has a variable baud rate. Baud rates can be generated by either Timer 1 or Timer 2 (available on the 83C51FA but not the 8051). Deciding which mode and timer to use is determined by the desired baud rate and clock frequency of the particular application.

Another consideration is the tolerance needed between serial ports. Since the serial port re-synchs its receiver at every start bit, only 8 or 9 bit-times are available to accumulate timing errors. As a result, the receiver and transmitter only have to be within about 5% of each other’s baud rate. Allowing equal error to both transmitter and receiver, only about 2% accuracy is actually needed.

Following is a discussion of both Modes 2 and 3 and examples of how to program each. The mode selection bits (SM0 and SM1) are located in SCON. The REN bit must also be set to enable reception.

<table>
<thead>
<tr>
<th>SM0</th>
<th>SM1</th>
<th>SM2</th>
<th>REN</th>
<th>TB8</th>
<th>RB8</th>
<th>TI</th>
<th>RI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BAUD RATE**
- Fosc/64 or
- Fosc/32
- Variable

**SERIAL PORT SET UP**

This section describes how to initialize the 83C51FA serial port for multiprocessor applications. Two different modes are available which provide 9-bit operation:
Example 1. Serial Port Mode 2

; Frequency = 12 MHz
; Desired Baud Rate = 375 kBaud ;
MOV SCON, #080H ; Serial port Mode 2 ;
ORL PCON, #08H ; Automatic Addressing (SM2 = 1),
; reception enabled (REN = 1)

Mode 2
Mode 2 uses a fixed baud rate of 1/32 or 1/64 of the oscillator frequency depending on the value of the SMOD1 bit in PCON. This mode basically offers a choice of two high-speed baud rates. With a 12 MHz clock frequency, baud rates of 187.5 kbaud or 375 kbaud can be obtained.

None of the timer/counters need to be set up for Mode 2. Only the SFRs SCON and PCON need to be defined.

PCON: Power Control Register (Not Bit Addressable)

<table>
<thead>
<tr>
<th>SMOD1</th>
<th>SMOD0</th>
<th>-</th>
<th>POF</th>
<th>GF1</th>
<th>GF0</th>
<th>PD</th>
<th>IDL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address = 87H

The baud rate in this mode is calculated by:

Mode 2 Baud Rate = \( \frac{2^{\text{SMOD1}} \times \text{Osc Freq}}{64} \)

SMOD1 = 0, Baud Rate = 1/64 Osc Freq
SMOD1 = 1, Baud Rate = 1/32 Osc Freq

Mode 3
Mode 3 of the serial port has a variable baud rate generated by either Timer 1 or Timer 2. The baud rate is generated by the rollover rate of the selected timer. The timer is operated in an auto-reload mode so it will roll over to the reload value selected in software.

Baud rates based off Timer 2 have less granularity so that almost any baud rate can be obtained at a given clock frequency. However, Timer 1 is sufficient if the desired baud rate can be obtained at the specified clock frequency. Remember baud rates only need about 2% accuracy.

Timer 1 Set Up

To generate baud rates Timer 1 is usually configured in 8-bit auto-reload mode (Mode 2). The mode select bits are M1 and M0 located in TMOD. To turn on Timer 1 the TR1 bit in TCON must be set. Also, the Timer 1 interrupt should be disabled in this application so that when the timer overflows it does not generate an interrupt.

TMOD: Timer/Counter Mode Control Register (Not bit addressable)

<table>
<thead>
<tr>
<th>GATE</th>
<th>C/T</th>
<th>M1</th>
<th>M0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address = 89H

TCON: Timer/Counter Control Register (Bit addressable)

<table>
<thead>
<tr>
<th>TF1</th>
<th>TR1</th>
<th>TF0</th>
<th>TR0</th>
<th>IE1</th>
<th>IT1</th>
<th>IE0</th>
<th>IT0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address = 88H

The formula for calculating the baud rate is given below. TH1 is the reload value for Timer 1 when it overflows.

Baud Rate = \( \frac{K \times \text{Osc Freq}}{32 \times 12 \times (256 - (\text{TH1}))} \)

K = 1 if SMOD1 = 0.
K = 2 if SMOD1 = 1. (SMOD1 is at PCON.7)

If the baud rate is known, the reload value TH1 can be calculated by:

\( \text{TH1} = 256 - \frac{K \times \text{Osc Freq}}{384 \times \text{Baud Rate}} \)

TH1 must be an integer value. Rounding off TH1 to the nearest integer may not produce the desired baud rate with the 2% accuracy required. In this case, another crystal frequency may have to be chosen.

Refer to Table 1 for timer reload values for commonly used baud rates.
Table 1. Commonly Used Baud Rates Generated by Timer 1

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>Osc Freq</th>
<th>SMOD1</th>
<th>Timer 1</th>
<th>Reload Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>TMOD</td>
<td></td>
</tr>
<tr>
<td>62.5K</td>
<td>12 MHz</td>
<td>1</td>
<td>20</td>
<td>FFH</td>
</tr>
<tr>
<td>19.2K</td>
<td>11.06 MHz</td>
<td>1</td>
<td>20</td>
<td>FDH</td>
</tr>
<tr>
<td>9.6K</td>
<td>11.06 MHz</td>
<td>0</td>
<td>20</td>
<td>FDH</td>
</tr>
<tr>
<td>4.8K</td>
<td>11.06 MHz</td>
<td>0</td>
<td>20</td>
<td>FAH</td>
</tr>
<tr>
<td>2.4K</td>
<td>11.06 MHz</td>
<td>0</td>
<td>20</td>
<td>F4H</td>
</tr>
<tr>
<td>1.2K</td>
<td>11.06 MHz</td>
<td>0</td>
<td>20</td>
<td>E8H</td>
</tr>
<tr>
<td>300</td>
<td>6 MHz</td>
<td>0</td>
<td>20</td>
<td>CCH</td>
</tr>
<tr>
<td>110</td>
<td>6 MHz</td>
<td>0</td>
<td>20</td>
<td>72H</td>
</tr>
</tbody>
</table>

Example 2. Serial Port Mode 3, with Timer 1 as Baud-Rate Generator

; Frequency = 11.0 MHz
; Desired Baud Rate = 19.2 kBaud
;
; TH1 = 256 - ((2) x (11.0 x 10^6))
; 
; (32) x (12) x (19200)

; = 253 = FDH

MOV SCON, #OFOH ; Serial port Mode 3, SM2 = 1,
; REN = 1
ORL PCON, #80H ; SMOD1 = 1
MOV TMOD, #20H ; Timer 1 Mode 2
MOV TH1, #OFDH ; Reload value for desired baud
; rate
SETB TR1 ; Turn on Timer 1

It can be seen that the exact frequency to generate the standard baud rates (19.2K, 9600, 4800, etc.) is 11.06 MHz. However, it is not necessary to use this exact frequency. With a 2% tolerance any crystal value from 10.8 MHz to 11.3 MHz is sufficient.

Timer 2 Set Up

Timer 2 has a special baud-rate generator mode which transmits and receives at the same baud rate. This mode is invoked by setting both the RCLK and TCLK bits in T2CON. To turn Timer 2 on the TR2 bit should also be set.

Unlike Timer 1, this mode does not require that the timer overflow interrupt be disabled. That is, when Timer 2 is in the baud-rate generator mode, its interrupt is disconnected from the Timer 2 overflow. This interrupt then becomes available as a third external interrupt. (For more information on external interrupts, refer to the chapter “Hardware Description of the 8051” in the Embedded Controller Handbook.)

T2CON: Timer/Counter 2 Control Register
(Bit Addressable)

<table>
<thead>
<tr>
<th>TF2</th>
<th>EXF2</th>
<th>RCLK</th>
<th>TCLK</th>
<th>EXEN2</th>
<th>TR2</th>
<th>C/T2</th>
<th>CP/RL2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address = C8H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This formula for calculating the baud rate is given below. (RCAP2H, RCAP2L) is the 16-bit reload value when Timer 2 overflows.

\[
\text{Baud Rate} = \frac{\text{Osc Freq}}{32 \times [65536 - (\text{RCAP2H}, \text{RCAP2L})]}
\]

where (RCAP2H, RCAP2L) is a 16-bit unsigned integer.
To obtain the reload value for RCAP2H and RCAP2L, the above equation can be rewritten as:

\[(RCAP2H, RCAP2L) = 65536 - \frac{\text{Osc Freq}}{32 \times \text{Baud Rate}}\]

Refer to Table 2 for reload values for commonly used baud rates.

Notice that when using Timer 2, most standard baud rates can be obtained at 12 MHz.

Table 2. Commonly Used Baud Rates Generated by Timer 2

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>Osc Freq</th>
<th>Timer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RCAP2H</td>
</tr>
<tr>
<td>375K</td>
<td>12 MHz</td>
<td>FF</td>
</tr>
<tr>
<td>9.6K</td>
<td>12 MHz</td>
<td>FF</td>
</tr>
<tr>
<td>4.8K</td>
<td>12 MHz</td>
<td>FF</td>
</tr>
<tr>
<td>2.4K</td>
<td>12 MHz</td>
<td>FF</td>
</tr>
<tr>
<td>1.2K</td>
<td>12 MHz</td>
<td>FE</td>
</tr>
<tr>
<td>300</td>
<td>12 MHz</td>
<td>FB</td>
</tr>
<tr>
<td>110</td>
<td>12 MHz</td>
<td>F2</td>
</tr>
<tr>
<td>300</td>
<td>6 MHz</td>
<td>FD</td>
</tr>
<tr>
<td>110</td>
<td>6 MHz</td>
<td>F9</td>
</tr>
</tbody>
</table>

Example 3. Serial Port Timer with Timer 2 as Baud-Rate Generator

; Frequency = 12 MHz
; Desired Baud Rate = 9600 Baud
; (RCAP2H, RCAP2L) = 65536 - \frac{12 \times 10^6}{32 \times (9600)}
; = 65497 = FFD9H

MOV SCON, #0F0H ; Serial port Mode 3, SM2 = 1, REN = 1
MOV RCAP2H, #0FFH ; Reload values for desired baud rate
MOV RCAP2L, #0D9H
MOV T2CON, #34H ; Timer 2 as baud rate generator, turn on Timer 2
Software Serial Port Implemented with the PCA

BETSY JONES
ECO APPLICATIONS ENGINEER
For microcontroller applications which require more than one serial port, the 83C51FA Programmable Counter Array (PCA) can implement additional half-duplex serial ports. If the on-chip UART is being used as an inter-processor link, the PCA can be used to interface the 83C51FA to additional asynchronous lines.

This application uses several different Compare/Capture modes available on the PCA to receive or transmit bytes of data. It is assumed the reader is familiar the PCA and ASM51. For more information on the PCA refer to the "Hardware Description of the 83C51FA" chapter in the Embedded Controller Handbook (Order No. 210918).

Introduction

The figure below shows the format of a standard 10-bit asynchronous frame: 1 start bit (0), 8 data bits, and 1 stop bit (1). The start bit is used to synchronize the receiver to the transmitter; at the leading edge of the start bit the receiver must set up its timing logic to sample the incoming line in the center of each bit. Following the start bit are eight data bits which are transmitted least significant bit first. The stop bit is set to the opposite state of the start bit to guarantee that the leading edge of the start bit will cause a transition on the line. It also provides a dead time on the line so that the receiver can maintain its synchronization.

Two of the Compare/Capture modes on the PCA are used in receiving and transmitting data bits. When receiving, the Negative-Edge Capture mode allows the PCA to detect the start bit. Then using the Software Timer mode, interrupts are generated to sample the incoming data bits. This same mode is used to clock out bits when transmitting.

This Application Note contains four sections of code:

1. List of variables
2. Initialization routine
3. Receive routine
4. Transmit routine

A complete listing of the routines and the test loop which was used to verify their operation is found in the Appendix. A total of three half-duplex channels were run at 2400 Baud in the test program. The listings shown here are simplified to one channel (Channel 0).

Variables

Listing 1 shows the variables used in both the receive and transmit routines. Flags are defined to signify the status of the reception or transmission of a byte (e.g. RCV_START_BIT, TXM_START_BIT). RCV_BUF and TXM_BUF simulate the on-chip serial port SBUF as two separate buffer registers. The temporary registers, RCV_REG and TXM_REG, are used to save bits as they are received or transmitted. Finally, two counter registers keep track of how many bits have been received or transmitted.

Variables are also needed to define one-half and one-full bit times in units of PCA timer ticks. (One bit time = 1 / baud rate.) With the PCA timer incremented every machine cycle, the equation to calculate one bit time can be written as:

\[
\text{Osc. Freq.} \times \frac{1}{12} \times \frac{1}{\text{baud rate}} = 1 \text{ bit time (in PCA timer ticks)}
\]

In this example, the baud rate is 2400 at 16 MHz.

\[
\frac{16 \text{ MHz}}{12} \times \frac{1}{2400} = 556 \text{ counts} = 22 \text{ Hex}
\]

The high and low byte of this value is placed in the variables FULL_BIT_HIGH and FULL_BIT_LOW, respectively. 115H is the value loaded into HALF_BIT_HIGH and HALF_BIT_LOW.
Listing 1. Variables used by the software serial port. Channel 0

```assembly
; Receive Routine
RCV_START_BIT_0 BIT 20H.0 ; Indicates start bit has been received
RCV_DONE_0 BIT 20H.1 ; Indicates data byte has been received
RCV_BUF_0 DATA 30H ; Software Receive "SBUF"
RCV_REG_0 DATA 31H ; Temporary register for receive bits
RCV_COUNT_0 DATA 32H ; Counter for receiving bits

; Transmit Routine:
TXM_START_BIT_0 BIT 20H.3 ; Indicates start bit has been transmitted
TXM_IN_PROG_0 BIT 20H.4 ; Indicates transmit is in progress
TXM_BUF_0 DATA 34H ; Software transmit "SBUF"
TXM_REG_0 DATA 35H ; Temporary register for transmitting bits
TXM_COUNT_0 DATA 36H ; Counter for transmitting bits
DATA_0 DATA 37H ; Register used for the test program

NEG_EDGE EQU 11H ; Two modes of operation
S_W_TIMER EQU 49H ; for compare/capture modules

HALF_BIT_HIGH EQU 01H ; Half bit time = 115H
HALF_BIT_LOW EQU 15H
FULL_BIT_HIGH EQU 02H ; Full bit time = 22CH
FULL_BIT_LOW EQU 2CH ; 2400 Baud at 16 MHz
```
Initialization

Listing 2 contains the initialization code for the receive and transmit process. Module 0 of the PCA is used as a receiver and is first set up to detect a negative edge from the start bit. Modules 2 and 3 are used for the additional 2 channels (see the Appendix). Module 3 is used as a separate software timer to transmit bits.

Listing 2. Initialization Routine

```
ORG 0000H
LJMP INITIALIZE
ORG 001BH
LJMP RECEIVE_DONE
ORG 0033H
LJMP RECEIVE

INITIALIZE: MOV SP, #5FH
INIT_PCA: MOV CMOD, #00H
          MOV CCON, #00H
          MOV CCAPM0, #NEG_EDGE
          MOV CCAPM3, #S_W_TIMER
          MOV CL, #00H
          MOV CH, #00H
          MOV IE, #0D8H
          SETB CR

; Timer 1 overflow -
; simulates "RI" interrupt
; PCA interrupt
; Initialize stack pointer
; (specific to test program)
; Increment PCA timer
; @ 1/12 Osc Frequency
; Clear all status flags
; Module 0 in negative-edge
; trigger mode (P1.3)
; Module 3 as software timer
; mode
; Init all needed interrupts
; EA, EC, ES, ET1
; Turn on PCA Counter
```

All flags and registers from Listing 1 should be cleared in the initialization process.

Receive Routine

Two operating modes of the PCA are needed to receive bits. The module must first be able to detect the leading edge of a start bit so it is initially set up to capture a 1-to-0 transition (i.e. Negative-Edge Capture mode). The module is then reconfigured as a software timer to cause an interrupt at the center of each bit to deserialize the incoming data. The flowchart for the receive routine is given in Figure 1.
Add 1 bit time to compare/capture registers

Reconfigure PCA module as a software timer

1 - 0 transition detected? YES

Reconfigure PCA module to capture 1 - 0 transitions

NO

Set valid start bit flag

Valid start bit received? YES

Valid stop bit received? YES

Start bit valid? NO

NO

Set receive done flag

Generate "RI" interrupt

Reconfigure PCA module to capture 1 - 0 transitions

NO

Add 1 bit time to compare/capture registers

Sample input pin

YES

RETI

RETI

Figure 1. Flowchart for the Receive Routine
Listing 3.1 shows the code needed to detect a start bit. Notice that the first software timer interrupt will occur one-half bit time after the leading edge of the start bit to check its validity. If it is valid, the RCV.START.BIT is set. The rest of the samples will occur a full bit time later. The RCV.COUNT register is loaded with a value of 9 which indicates the number of bits to be sampled: 8 data bits and 1 stop bit.

**Listing 3.1. Receive Interrupt Routine**

```
RECEIVE:  PUSH ACC
          PUSH PSW
;
MODULE_0:  CLR CCF0 ; Assume reception on
          ; Module 0
      MOV A, CCAPM0 ; Check mode of module. If
      ANL A, #0111111B ; set up to receive negative
      CJNE A, #NEG_EDGE, RCV.START_0 ; edges, then module
          ; is waiting for a start bit
;
      CLR C ; Update compare/capture
      MOV A, #HALF_BIT_LOW ; registers for half bit time
      ADD A, CCAPOL
      MOV CCAPOL, A ; Half bit time = 115H
      MOV A, #HALF_BIT_HIGH
      ADDC A, CCAPOL
      MOV CCAPOL, A
      MOV CCAPMO, #S_W_TIMER ; Reconfigure module 0 as
          ; a software timer to sample
      POP PSW ; bits
      POP ACC
      RETI
;
RCV_START_0: CJNE A, #S_W_TIMER, ERROR_0 ; Check module is
          ; configured as a software
          ; timer, otherwise error.
      JB RCV.START_BIT_0, RCV.BYTE_0 ; Check if start bit
          ; is received yet.
      JB P1.3, ERROR_0 ; Check that start bit = 0,
          ; otherwise error.
      SETB RCV.START_BIT_0 ; Signify valid start bit
          ; was received
      MOV RCV.COUNT_0, #09H ; Start counting bits sampled
;
      CLR C ; Update compare/capture
      MOV A, #FULL_BIT_LOW ; registers to sample
      ADD A, CCAPOL
      MOV CCAPOL, A ; incoming bits
      MOV A, #FULL_BIT_HIGH
      ADDC A, CCAPOL
      MOV CCAPOL, A
      POP PSW
      POP ACC
      RETI
```
The next 8 timer interrupts will receive the incoming data bits; the RCV\_COUNT register keeps track of how many bits have been sampled. As each bit is sampled, it is shifted through the Carry Flag and saved in RCV\_REG. The ninth sample checks the validity of the stop bit. If it is valid, the data byte is moved into RCV\_BUF.

The main routine must have a way to know that a byte has been received. With the on-chip UART, the RI (Receive Interrupt) bit is set whenever a byte has been received. For the software serial port, any unimplemented interrupt vector can be used to generate an interrupt when a byte has been received. This routine uses the Timer 1 Overflow interrupt (its selection is arbitrary). A routine to test this interrupt is included in the listing in the Appendix.

Listing 3.2. Receive Interrupt Routine (Continued)

```assembly
RCV\_BYTE\_0: DJNZ RCV\_COUNT\_0, RCV\_DATA\_0 ; On 9th sample,  
             ; check for valid stop bit
RCV\_STOP\_0: JNB P1.3, ERROR\_0  
       MOV RCV\_BUF\_0, RCV\_REG\_0 ; Save received byte in  
                     ; receive "SBUF"  
       SETB RCV\_DONE\_0  
                     ; Flag which module received  
       SETB TF1  
                     ; Generate an interrupt so  
                     ; main program knows a byte  
                     ; has been received  
                     ; (Note: selection of TF1 is  
                     ; arbitrary)  
       MOV CCAPM0, \#NEG\_EDGE  
                     ; Reconfigure module 0 for  
                     ; Reception of a start bit
POP PSW  
POP ACC  
RETI

RCV\_DATA\_0: MOV C, P1.3  
       MOV A, RCV\_REG\_0  
       RRC A  
       MOV RCV\_REG\_0, A  
                     ; Sampling data bits  
                     ; Shifts bits thru CY into  
                     ; ACC  
                     ; Save each reception in  
                     ; temporary register  
                     ; Update c/c register for  
                     ; next sample time
CLR C  
       MOV A, \#FULL\_BIT\_LOW  
       ADD A, CCAPOL  
       MOV CCAPOL, A  
       MOV A, \#FULL\_BIT\_HIGH  
       ADDC A, CCAPOH  
       MOV CCAPOH, A  
POP PSW  
POP ACC  
RETI

In addition, an error routine (Listing 3.3) is included for invalid start or stop bits to offer some protection against noise. If an error occurs, the module is re-initialized to look for another start bit.

Listing 3.3 Error Routine for Receive Routine

```assembly
ERROR\_0: MOV CCAPM0, \#NEG\_EDGE ; Reset module to look for  
                     ; start bit
CLR RCV\_START\_BIT\_0 ; Clear flags which might  
                     ; have been set
POP PSW  
POP ACC  
RETI
```

2-227
Transmit Routine

Another PCA module is configured as a software timer to interrupt the CPU every bit time. With each timer interrupt one or more bits can be transmitted through port pins. In the test program three channels were operated simultaneously, but in the listings below, one channel is shown for simplicity. The selection of port pins is user programmable. The flowchart for the transmit routine is given in Figure 2.

![Figure 2. Flowchart for the Transmit Routine](image)

When a byte is ready to be transmitted, the main program moves the data byte into the TXM_BUF register and sets the corresponding TXM_IN_PROGRESS bit. This bit informs the interrupt routine which channel is transmitting. The data byte is then moved in the storage register TXM_REG, and the TXM_COUNT is loaded. This main routine is shown in Listing 4.1.

Listing 4.1 Transmit Set Up Routine. Channel 0.

```
TXM_ON_0: CLR TXM_START_BIT_0 ; Clear status flag from previous transmission
MOV TXM_BUF_0, DATA_0 ; Load "SBUF" with data byte
MOV TXM_REG_0, TXM_BUF_0
MOV TXM_COUNT_0, #09 ; 8 data bits + 1 stop bit
SETB TXM_IN_PROGRESS_0
```

2-228
Listing 4.2 shows the transmit interrupt routine. The first time through, the start bit is transmitted. As each successive interrupt outputs a bit, the contents of TXM_REG is shifted right one place into the Carry flag, and the TXM_COUNT is decremented. When TXM_COUNT equals zero, the stop bit is transmitted.

### Listing 4.2. Transmit Interrupt Routine

```assembly
TRANSMIT: PUSH ACC
PUSH PSW
CLR CCF3 ; Clear s/w timer interrupt
; for transmitting bits
JNB TXM_IN_PROGRESS_0, TRANSmit_1 ; Check which
; channel is transmitting.
; "TRANSMIT_1" is listed in
; the Appendix

; TRANSmit_0: JB TXM_START_BIT_0, TXM_BYTE_0 ; If start bit
; has been sent, continue
; transmitting bits.
; Otherwise transmit start
; bit
; Signify start bit sent
CLR P3.2
SETB TXM_START_BIT_0
JMP TXM_EXIT

; TXM_BYTE_0: DJNZ TXM_COUNT_0, TXM_DATA_0 ; If bit count
; equals 1 thru 9, transmit
; data bits (8 total)

; TXM_STOP_0: SETB P3.2
; When bit count = 0,
; transmit stop bit
; Indicate transmission is
; finished and ready for
; next byte
CLR TXM_IN_PROGRESS_0
JMP TXM_EXIT

; TXM_DATA_0: MOV A, TXM_REG_0
; RRC A
; MOV P3.2, C
; MOV TXM_REG_0, A
; Transmit one bit at a time
; through the carry bit
; Save what’s not been sent

; TXM_EXIT: CLR C
MOV A, #FULL_BIT_LOW
ADD A, CCAP3L
MOV CCAP3L, A
MOV A, #FULL_BIT_HIGH
ADDCC A, CCAP3H
MOV CCAP3H, A
POP PSW
POP ACC
RET
```

### Conclusion

The software routines in the Appendix can be altered to vary the baud rate and number of channels to fit a particular application. The number of channels which can be implemented is limited by the CPU time required to service the PCA interrupt. At higher baud rates, fewer channels can be run.

The test program verifies the simultaneous operation of three half-duplex channels at 2400 Baud and the on-chip full-duplex channel at 9600 Baud. Thirty-three percent of the CPU time is required to operate all four channels. The test was run for several hours with no apparent malfunctions.
This program tests the receive routines of a software serial port.

Three half-duplex channels are implemented in software to run at
2400 baud (16MHz). The on-chip serial port is also running full-duplex
at 9600 Baud. Thirty-three percent of the CPU time is required to run
all four ports simultaneously.

To test the receive routines, "dummy" terminals transmit 00 - FF hex
continually to the PCA. When the first byte is received, it is
compared with 00. If the comparison is valid, the compare value is
incremented and the routine waits to receive the next byte. Error
routines toggle various Port 3 pins if an invalid comparison occurs
or if an invalid start bit or stop bit is received.

    0000 0000 020036
    0000 0011 001B 02025C
    0000 0023 0023 020282
    0000 0033 0204D3

    ORG 01H
    ORG 01B
    ORG 0000
    ORG 001B
    ORG 0023H
    ORG 0024H
    ORG 0025H
    ORG 0026H

    0000 0000 001B 0204D3
    0000 0023 0023 0204D3

    JMP INITIALIZE
    JMP RECEIVE_DONE ; Timer 1 Overflow - simulates "RI" interrupt
    JMP SERIAL_PORT ; Serial port interrupt
    JMP RECEIVE ; PCA interrupt

---

VARIABLES USED BY THE SOFTWARE SERIAL PORT RECEIVE ROUTINE:

- RCV_START_BIT_0 : Indicates start bit has been received
- RCV_START_BIT_1 : Indicates start bit has been received
- RCV_DONE : Indicates data byte has been received
- RCV_DONE_1 : Indicates data byte has been received
- RCV_DONE_2 : Indicates data byte has been received
- RCV_ON_0 : Used in main test program to check for a received byte
- RCV_ON_1 : Used in main test program to check for a received byte
- RCV_ON_2 : Used in main test program to check for a received byte
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
</table>
| 0030  | 199 | RCV_BF_0 | DATA | 30H | ; Software receive *SBUF*
| 0030  | 200 | RCV_BF_1 | DATA | 40H | |
| 0031  | 201 | RCV_Reg_0 | DATA | 31H | ; Temporary register for |
| 0041  | 204 | RCV_Reg_1 | DATA | 41H | receiving bits |
| 0051  | 205 | RCV_Reg_2 | DATA | 51H | |
| 0052  | 209 | RCV_Count_0 | DATA | 32H | ; Counter for receiving bits |
| 0053  | 210 | RCV_Count_1 | DATA | 42H | |
| 0053  | 210 | RCV_Count_2 | DATA | 52H | |
| 0011  | 213 | REG_EDGE | EQU | 11H | ; Two modes of operation for the |
| 0049  | 216 | SW TIMER | EQU | 49H | ; Compare/Capture modules |
| 0051  | 218 | RCV_COUNT | DATA | 33H | ; Used in test program to check |
| 0053  | 213 | COUNT_1 | DATA | 43H | bytes being received |
| 0053  | 213 | COUNT_2 | DATA | 53H | |
| 0032  | 207 | RCV_COUNT_0 | DATA | 32H | |
| 0042  | 208 | RCV_COUNT_1 | DATA | 42H | |
| 0052  | 209 | RCV_COUNT_2 | DATA | 52H | |

**INITIALIZATION ROUTINE**

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0036</td>
<td>75815F</td>
<td>230</td>
<td>INITIALIZE: MOV SP, #5FH</td>
</tr>
<tr>
<td>0039</td>
<td>759000</td>
<td>232</td>
<td>INIT_FC: MOV CMD, #00H</td>
</tr>
<tr>
<td>003C</td>
<td>75A000</td>
<td>233</td>
<td>MOV CCME, #00H</td>
</tr>
<tr>
<td>003F</td>
<td>75A111</td>
<td>234</td>
<td>MOV CCPM0, #NEG EDGE</td>
</tr>
<tr>
<td>0042</td>
<td>75B011</td>
<td>235</td>
<td>MOV CCPM0, #NEG EDGE</td>
</tr>
<tr>
<td>0043</td>
<td>75C011</td>
<td>236</td>
<td>MOV CCPM0, #NEG EDGE</td>
</tr>
<tr>
<td>0048</td>
<td>75E900</td>
<td>238</td>
<td>MOV CL, #00H</td>
</tr>
<tr>
<td>004A</td>
<td>75F900</td>
<td>239</td>
<td>MOV CH, #00H</td>
</tr>
<tr>
<td>004E</td>
<td>75ABDB</td>
<td>240</td>
<td>MOV IE, #00H</td>
</tr>
<tr>
<td>0051</td>
<td>D200</td>
<td>241</td>
<td>SETB CR</td>
</tr>
<tr>
<td>0053</td>
<td>759850</td>
<td>243</td>
<td>INIT_SP: MOV SCON, #50H</td>
</tr>
<tr>
<td>0056</td>
<td>75CB00</td>
<td>244</td>
<td>MOV RCAPH, #0FFH</td>
</tr>
<tr>
<td>0059</td>
<td>75CC00</td>
<td>245</td>
<td>MOV RCAPL, #00CH</td>
</tr>
<tr>
<td>005C</td>
<td>75C334</td>
<td>246</td>
<td>MOV T2CON, #34H</td>
</tr>
<tr>
<td>005F</td>
<td>C200</td>
<td>249</td>
<td>INITIAL_FLAGS: CLR RCV_START_BIT_0</td>
</tr>
<tr>
<td>0061</td>
<td>C208</td>
<td>250</td>
<td>CLR RCV_START_BIT_1</td>
</tr>
<tr>
<td>0063</td>
<td>C210</td>
<td>251</td>
<td>CLR RCV_START_BIT_2</td>
</tr>
<tr>
<td>0065</td>
<td>C201</td>
<td>253</td>
<td>CLR RCV DONE_0</td>
</tr>
</tbody>
</table>

**270531-12**
Port 3 pins used in test program for error routines

Main program:

Interrupt routines:

Main program continually checks each channel for a received byte.

Once a byte is received, it is compared with the current value in the "COUNT" register.

Main test routine - receive bits
Error in comparison on module 0
Discontinue receiving bytes

Error in comparison on module 1

Error in comparison on module 2

PCA INTERRUPT ROUTINE - RECEIVE BITS

Check which module caused PCA interrupt and jump to appropriate routine

Reception on module 0
Check mode of module. If set up to receive negative edges, then module is waiting for a start bit

Update Compare/Capture registers for half a bit time

Configure module 0 as a software timer to sample bits
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>32</td>
<td>364</td>
<td>RETI</td>
</tr>
<tr>
<td>0112</td>
<td>B4494B</td>
<td>366</td>
<td>RCV_START_0: CJE A, RCV_TIMER, ERROR_0; Check module is configured as a software timer, otherwise error.</td>
</tr>
<tr>
<td>0115</td>
<td>20001A</td>
<td>367</td>
<td>JB RCV_START_BIT_0, RCV_BYTE_0; Check if start bit has been received yet.</td>
</tr>
<tr>
<td>0118</td>
<td>209345</td>
<td>368</td>
<td>JB P1.3, ERROR_0; Check that start bit = 0, otherwise error.</td>
</tr>
<tr>
<td>011B</td>
<td>D200</td>
<td>369</td>
<td>SETB RCV_START_BIT_0; Signify valid start bit was received.</td>
</tr>
<tr>
<td>011D</td>
<td>753209</td>
<td>370</td>
<td>MOV RCV_COUNT_0, 09H; Start counting bits sampled.</td>
</tr>
<tr>
<td>0120</td>
<td>C3</td>
<td>371</td>
<td>CLR C; Update C/C registers to sample incoming bits; Full bit time = 22CH</td>
</tr>
<tr>
<td>0121</td>
<td>742C</td>
<td>372</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>0122</td>
<td>25EA</td>
<td>373</td>
<td>ADD A, CCAPBE</td>
</tr>
<tr>
<td>0123</td>
<td>F5EA</td>
<td>374</td>
<td>MOV CCAPBE_A,</td>
</tr>
<tr>
<td>0124</td>
<td>1402</td>
<td>375</td>
<td>POP PSI</td>
</tr>
<tr>
<td>0125</td>
<td>25EA</td>
<td>376</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>0126</td>
<td>C3</td>
<td>377</td>
<td>CLR C; Update C/C registers to sample incoming bits; Full bit time = 22CH</td>
</tr>
<tr>
<td>0127</td>
<td>D200</td>
<td>378</td>
<td>SETB RCV_BYTE_0; On 9th sample, check for valid stop bit.</td>
</tr>
<tr>
<td>0128</td>
<td>25EA</td>
<td>379</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>0129</td>
<td>C3</td>
<td>380</td>
<td>CLR C; Update C/C registers to sample incoming bits; Full bit time = 22CH</td>
</tr>
<tr>
<td>012A</td>
<td>D200</td>
<td>381</td>
<td>MOV CCAPBE_A,</td>
</tr>
<tr>
<td>012B</td>
<td>25EA</td>
<td>382</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>012C</td>
<td>C3</td>
<td>383</td>
<td>CLR C; Update C/C registers to sample incoming bits; Full bit time = 22CH</td>
</tr>
<tr>
<td>012D</td>
<td>D200</td>
<td>384</td>
<td>MOV CCAPBE_A,</td>
</tr>
<tr>
<td>012E</td>
<td>25EA</td>
<td>385</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>012F</td>
<td>C3</td>
<td>386</td>
<td>CLR C; Update C/C registers to sample incoming bits; Full bit time = 22CH</td>
</tr>
<tr>
<td>0130</td>
<td>1402</td>
<td>387</td>
<td>MOV PSI</td>
</tr>
<tr>
<td>0131</td>
<td>742C</td>
<td>388</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>0132</td>
<td>25EA</td>
<td>389</td>
<td>ADD A, CCAPBE</td>
</tr>
<tr>
<td>0133</td>
<td>F5EA</td>
<td>390</td>
<td>MOV CCAPBE_A,</td>
</tr>
<tr>
<td>0134</td>
<td>1402</td>
<td>391</td>
<td>POP PSI</td>
</tr>
<tr>
<td>0135</td>
<td>742C</td>
<td>392</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>0136</td>
<td>25EA</td>
<td>393</td>
<td>ADD A, CCAPBE</td>
</tr>
<tr>
<td>0137</td>
<td>F5EA</td>
<td>394</td>
<td>MOV CCAPBE_A,</td>
</tr>
<tr>
<td>0138</td>
<td>1402</td>
<td>395</td>
<td>POP PSI</td>
</tr>
<tr>
<td>0139</td>
<td>25EA</td>
<td>396</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>013A</td>
<td>D200</td>
<td>397</td>
<td>POP ACC</td>
</tr>
<tr>
<td>013B</td>
<td>25EA</td>
<td>398</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>013C</td>
<td>C3</td>
<td>399</td>
<td>CLR C; Update C/C registers to sample incoming bits; Full bit time = 22CH</td>
</tr>
<tr>
<td>013D</td>
<td>D200</td>
<td>400</td>
<td>MOV CCAPBE_A,</td>
</tr>
<tr>
<td>013E</td>
<td>25EA</td>
<td>401</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>013F</td>
<td>C3</td>
<td>402</td>
<td>CLR C; Update C/C registers to sample incoming bits; Full bit time = 22CH</td>
</tr>
<tr>
<td>0140</td>
<td>1402</td>
<td>403</td>
<td>POP PSI</td>
</tr>
<tr>
<td>0141</td>
<td>25EA</td>
<td>404</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>0142</td>
<td>D200</td>
<td>405</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0143</td>
<td>25EA</td>
<td>406</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>0144</td>
<td>D200</td>
<td>407</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0145</td>
<td>25EA</td>
<td>408</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>0146</td>
<td>D200</td>
<td>409</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0147</td>
<td>25EA</td>
<td>410</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>0148</td>
<td>D200</td>
<td>411</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0149</td>
<td>25EA</td>
<td>412</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>014A</td>
<td>D200</td>
<td>413</td>
<td>POP ACC</td>
</tr>
<tr>
<td>014B</td>
<td>25EA</td>
<td>414</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>014C</td>
<td>D200</td>
<td>415</td>
<td>POP ACC</td>
</tr>
<tr>
<td>014D</td>
<td>25EA</td>
<td>416</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>014E</td>
<td>D200</td>
<td>417</td>
<td>POP ACC</td>
</tr>
<tr>
<td>014F</td>
<td>25EA</td>
<td>418</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>0150</td>
<td>D200</td>
<td>419</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0151</td>
<td>25EA</td>
<td>420</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>0152</td>
<td>D200</td>
<td>421</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0153</td>
<td>25EA</td>
<td>422</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>0154</td>
<td>D200</td>
<td>423</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0155</td>
<td>25EA</td>
<td>424</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>0156</td>
<td>D200</td>
<td>425</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0157</td>
<td>25EA</td>
<td>426</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>0158</td>
<td>D200</td>
<td>427</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0159</td>
<td>25EA</td>
<td>428</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>015A</td>
<td>D200</td>
<td>429</td>
<td>POP ACC</td>
</tr>
<tr>
<td>015B</td>
<td>25EA</td>
<td>430</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
<tr>
<td>015C</td>
<td>D200</td>
<td>431</td>
<td>POP ACC</td>
</tr>
<tr>
<td>015D</td>
<td>25EA</td>
<td>432</td>
<td>MOV A, FULL_BIT_LOW</td>
</tr>
<tr>
<td>015E</td>
<td>D200</td>
<td>433</td>
<td>POP ACC</td>
</tr>
<tr>
<td>015F</td>
<td>25EA</td>
<td>434</td>
<td>MOV A, FULL_BIT_HIGH</td>
</tr>
</tbody>
</table>
| 0160 | C285 | 435 | CLR P3.5; Error routine for invalid start or stop bit or invalid mode comparison.
IICS-51 KACRO ASSEMBLER

0181 0208 450 SETB Rev
0183 154209 11111
0185 142C 454 MOV
0187 25EB 455 ADD
0189 25EB 456 ADD
018B 154209 11111
018D 750B11 460 MOV
018F 0D00 461 POP
0191 0D00 462 POP
0193 0D00 463 POP
0195 D542l2 464 DJNZ RCV_COUNT_1, RCV_DATA_1
0196 D542l2 465 DJNZ RCV_COUNT_1, RCV_DATA_1
0197 300428 466 DJNZ RCV_COUNT_1, RCV_DATA_1
0199 85414D 467 MOV RCV_BUF, RCV_REG_1
019B D209 468 SETB RCV_DONE_1
019D D209 469 SETB RCV_DONE_1
019F 750B11 470 MOV CCAFL, #NEG_EDGE
01A1 0D00 471 POP
01A3 0D00 472 POP

; Port pin used for debug only
; Clear flags which might have been set

; Reset module to look for start bit

; Similar to module 0

DIM

2255

270531-16
MCS-51 MACRO ASSEMBLER

LOC OBJ | OBJ | LINE | SOURCE
------- | ---- | ---- | ----
01BE 32 | 473 | | RETI
01BF A294 | 474 | MOV C, Pl.4
01C1 Z41 | 475 | MOV A, RCV_REG_1
01C4 F341 | 476 | RRC A
01C5 32 | 477 | MOV RCV_REG_1, A
01C7 413 | 478 | ;
01C8 474 | 479 | ;
01C9 25EB | 480 | CLR C
01CA F3EB | 481 | MOV A, #FULL_BIT_LOW
01CD 7402 | 482 | ADD A, CCAP1L
01CF 3F9B | 483 | MOV CCAP1L, A
01D1 F5FB | 484 | MOV CCAP1H, A
01D3 D06 | 485 | POP PSW
01D4 D0E | 486 | POP ACC
01D7 32 | 487 | RETI
01D8 CB86 | 488 | ;
01DA 74D11 | 489 | MOV CCAP2L, #NEG_EDGE
01DB C2B8 | 490 | CLR RCV_START_BIT_1
01DF D06 | 491 | POP PSW
01E0 D0E | 492 | POP ACC
01E3 32 | 493 | RETI
01E4 C2DA | 494 | ;
01E5 E1DC | 495 | ------
01E6 547F | 496 | ; Similar to module 0
01E7 B411 | 497 | MODULE_2:
01E8 5415 | 498 | CLR CC2
01E9 546 | 499 | MOV A, CCAP2H
01EA 547 | 500 | ANL A, #0111111B
01EB 54F | 501 | CJNE A, #NEG_EDGE, RCV_START_2
01ED 32 | 502 | ;
01EE 7415 | 503 | CLR C
01F0 25EC | 504 | MOV A, #FULL_BIT_LOW
01F1 F3EB | 505 | ADD A, CCAP1L
01F4 7402 | 506 | MOV CCAP1L, A
01F5 F5FC | 507 | MOV CCAP1H, A
01F6 74DC49 | 508 | MOV CCAP2L, #6_W_TIMER
01F7 D0D0 | 509 | POP PSW
01F8 D0E | 510 | POP ACC
0201 32 | 511 | RETI
0202 B4494B | 512 | ;
0203 20181A | 513 | RCV_START_2:
0204 209545 | 514 | CJNE A, #6_W_TIMER, ERROR_2
0205 D210 | 515 | JB RCV_START_BIT_2, RCV_BYTE_2
0206 D255 | 516 | JB Pl.5, ERROR_2
0207 D75209 | 517 | ;
0208 D7520A | 518 | MOV RCV_COUNT_2, #09H

01/01/80 PAGE 7

270531-17
This routine simulates the "RI" interrupt. When a byte is received on one of the channels, this interrupt is generated. Bits are set so the main routine knows which channel received a byte.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0262</td>
<td>390106</td>
<td>583</td>
<td>JNB RCV DONE 0, RCV_1</td>
</tr>
<tr>
<td>0265</td>
<td>C921</td>
<td>584</td>
<td>CLR RCV DONE 0</td>
</tr>
<tr>
<td>0267</td>
<td>C200</td>
<td>585</td>
<td>CLR RCV START BIT 0</td>
</tr>
<tr>
<td>0269</td>
<td>0203</td>
<td>586</td>
<td>SETB RCV ON 0</td>
</tr>
<tr>
<td>0268</td>
<td>390906</td>
<td>588</td>
<td>RCV 1:</td>
</tr>
<tr>
<td>0266</td>
<td>C929</td>
<td>589</td>
<td>CLR RCV DONE 1, RCV 2</td>
</tr>
<tr>
<td>0270</td>
<td>C208</td>
<td>590</td>
<td>CLR RCV START BIT 1</td>
</tr>
<tr>
<td>0272</td>
<td>D20A</td>
<td>591</td>
<td>SETB RCV ON 1</td>
</tr>
<tr>
<td>0274</td>
<td>391106</td>
<td>593</td>
<td>RCV 2:</td>
</tr>
<tr>
<td>0277</td>
<td>C211</td>
<td>594</td>
<td>CLR RCV DONE 2</td>
</tr>
<tr>
<td>0279</td>
<td>C210</td>
<td>595</td>
<td>CLR RCV START BIT 2</td>
</tr>
<tr>
<td>027B</td>
<td>D212</td>
<td>596</td>
<td>SETB RCV ON 2</td>
</tr>
<tr>
<td>027D</td>
<td>D000</td>
<td>598</td>
<td>RETURN:</td>
</tr>
<tr>
<td>027F</td>
<td>D000</td>
<td>599</td>
<td>POP PSW</td>
</tr>
<tr>
<td>0281</td>
<td>32</td>
<td>600</td>
<td>RETI</td>
</tr>
<tr>
<td>0282</td>
<td>C008</td>
<td>613</td>
<td>SERIAL PORT:</td>
</tr>
<tr>
<td>0284</td>
<td>C008</td>
<td>614</td>
<td>PUSH ACC</td>
</tr>
<tr>
<td>0286</td>
<td>30980B</td>
<td>615</td>
<td>JNB RI, TXM</td>
</tr>
<tr>
<td>0289</td>
<td>E599</td>
<td>616</td>
<td>MOV A, SBUF</td>
</tr>
<tr>
<td>028B</td>
<td>C794</td>
<td>617</td>
<td>CLR RI</td>
</tr>
<tr>
<td>028D</td>
<td>F599</td>
<td>618</td>
<td>MOV SBUF, A</td>
</tr>
<tr>
<td>028F</td>
<td>D000</td>
<td>619</td>
<td>POP PSW</td>
</tr>
<tr>
<td>0291</td>
<td>D000</td>
<td>620</td>
<td>POP ACC</td>
</tr>
<tr>
<td>0293</td>
<td>32</td>
<td>621</td>
<td>RETI</td>
</tr>
<tr>
<td>0294</td>
<td>C929</td>
<td>622</td>
<td>TXM:</td>
</tr>
<tr>
<td>0296</td>
<td>D000</td>
<td>623</td>
<td>CLR TI</td>
</tr>
<tr>
<td>0298</td>
<td>D000</td>
<td>624</td>
<td>POP PSW</td>
</tr>
<tr>
<td>029A</td>
<td>D000</td>
<td>625</td>
<td>POP ACC</td>
</tr>
<tr>
<td>029C</td>
<td>32</td>
<td>626</td>
<td>RETI</td>
</tr>
<tr>
<td>029E</td>
<td>32</td>
<td>627</td>
<td>END</td>
</tr>
</tbody>
</table>

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
This program tests the transmit routines for the software serial port.

To initialize the first transmission, the compare values are loaded before
the timer is started. Successive interrupts are generated every bit
by the software timer.

For test purposes, the data transmitted increments from 00 to FF hex.
"Dummy" terminals receive these bytes and display the bytes as they
are incremented.

- Dummy terminals receive these bytes and display the bytes as they
are incremented.

**VARIABLES USED BY THE SOFTWARE SERIAL PORT**

- **TXM.START.BIT.0**: Indicates start bit has been transmitted
- **TXM.START.BIT.1**: Indicates transmit is in progress
- **TXM.START.BIT.2**: Temporary register for transmitting bits
- **TXM.START.BIT.3**: Register used for the test program
; Software timer mode for the
; compare/capture module
; Full bit time = 22CH
; Software timer mode for the
; compare/capture module
; Full bit time = 22CH
; 2400 baud at 16 MHz

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
; turn Timer 2 on

; Increment PCA timer @ 1/12 osc. freq.
; Clear all status flags

; Module 3 configured as software timer
; Initialize all needed interrupts

; Serial port in mode 1 (8-bit UART)
; Reload values for 9600 Baud @ 16 MHz
; Timer 2 as a baud-rate generator,
MCS-51 MACRO ASSEMBLER

008F 02000D
0092 300408
0095 30016
0098 30124
009B 80F5
009D C023
009F 0537
00A1 85374
00A4 853435
00A7 753609
00AA 80E4
00A8 8047
00B0 8547
00B2 854744
00B5 854445
00B8 754609
00BB 820C
00BD 8003
00BF C213
00C1 8547
00C3 854744
00C6 854445
00C9 754609
00CC 8214
00CE 8042
00D0 C0E0
00D2 C0D0
00D4 C20B
00D6 30041E
00D9 200307
00DC C262
00DE 8253
00E0 8200F7

01/01/80 PAGE 3

MAIN TEST ROUTINE - TRANSMIT BITS

;--------------------------------------------------

; First TXM

JMP TXM_ON_0

; Main TXM

JNB TXM IN PROGRESS_0, TXM ON_0 ; Determine if ready to send
INC DATA_0 ; transmission
MOV TXM BUF_0, DATA_0 ; Load *SBUR* with data byte
MOV TXM REG_0, TXM BUF_0
MOV TXM COUNT_0, $3FH
; 8 data bits + 1 stop bit
SETB TXM IN PROGRESS_0
JMP MAIN_TXM

; TXM ON_1

CLR TXM START_BIT_0 ; Clear flag from previous
INC DATA_0
MOV TXM BUF_1, DATA_1
MOV TXM REG_1, TXM BUF_1
MOV TXM COUNT_1, $3FH
SETB TXM IN PROGRESS_1
JMP MAIN_TXM

; TXM ON_2

CLR TXM START_BIT_2
INC DATA_2
MOV TXM BUF_2, DATA_2
MOV TXM REG_2, TXM BUF_2
MOV TXM COUNT_2, $3FH
SETB TXM IN PROGRESS_2
JMP MAIN_TXM

; PCA INTERRUPT ROUTINE - TRANSMIT BITS

;--------------------------------------------------

PUSH ACC
PUSH PSW
CLR CCF3
JNB TXM IN PROGRESS_0, TXM_BYTE_0 ; Check which channel is
; transmitting
JMP TXM_BYTE_0

; CHANNEL 0

JB TXM_START_BIT_0, TXM_BYTE_0 ; If start bit has been sent,
; continue transmitting data bits,
CLR P3.2 ; otherwise transmit start bit
SETB TXM START_BIT_1 ; Signify start bit sent
; Check next transmit pin

270531-22
LOC OBJ LINE SOURCE
00E3 D53607 0089 TXM_BYTE_0: DJNZ TXM_COUNT_0, TXM_DATA_0 ; If bit count equals 1 thru 9,
00E6 D2B2 0101 TXM_STOP_0: SETB P3.2 ; Transmit data bits (8 total)
00EA C204 0102 CLR TXM_IN_PROGRESS_0 ; When bit count = 0, transmit stop bit
00E9 0200F7 0103 JMP TRANSMIT_1 ; Indicate transmission is finished and
00EE E535 0104 TXM_DATA_0: MOV A, TXM_REG_0 ; ready for next byte
00F0 311 TXM_STOP_0: MOV P3.1, C ; Check next transmit pin
00F2 0200F7 0105 ; ; CHANNEL 1
00F4 311 TXM_BYTE_1: DJNZ TXM_COUNT_1, TXM_DATA_1 ; Transmit one bit at a time
00F6 D2B3 0106 TXM_STOP_1: SETB P3.3 ; Through the carry bit
00F8 020118 0107 JMP TRANSMIT_2 ; Save what's not been sent
00F9 331 TXM_DATA_1: MOV A, TXM_REG_1 ; ; Check next transmit pin
00FB E545 0108 ; ; CHANNEL 2
00FC 311 TXM_BYTE_2: DJNZ TXM_COUNT_2, TXM_DATA_2 ; Similar to TRANSMIT_0
00FE D2B3 0109 TXM_STOP_2: SETB P3.4 ; Similar to TRANSMIT_0
0101 020118 0110 JMP TRANSMIT_2 ; Similar to TRANSMIT_O
0103 D545 0111 TXM_DATA_2: MOV A, TXM_REG_2
0105 331 ; ; CHANNEL 1
0107 D2B3 0112 TXM_STOP_1: SETB P3.3
0109 C30C 0113 CLR TXM_IN_PROGRESS_1
010B 020118 0114 JMP TRANSMIT_2
010D E545 0115 TXM_DATA_1: MOV A, TXM_REG_1
010F 331 ; ; CHANNEL 2
0111 392B3 0116 MOV P3.2, C
0113 F545 0117 MOV TXM_REG_0, A
0115 020118 0118 ; ; CHANNEL 1
0117 30141E 0119 SETB TXM_START_BIT_1
011B 020118 0120 JB TXM_START_BIT_1 ; TXM_BYTE_1
011D C284 0121 TXM_BYTE_1: CLR P3.3
011F 020113 0122 SETB TXM_START_BIT_2
0121 D213 0123 JMP TXM_EXIT
0123 D55607 0124 TXM_COUNT_2, TXM_DATA_2
0125 331 ; ; CHANNEL 2
0127 D2B4 0128 TXM_STOP_2: SETB P3.4
0129 C314 0130 CLR TXM_IN_PROGRESS_2
012B 020139 0131 JMP TXM_EXIT
012D E555 0132 TXM_DATA_2: MOV A, TXM_REG_2
0131 392B4 0133 MOV P3.2, C
0133 F555 0134 MOV TXM_REG_0, A
0135 020139 0136 JMP TXM_EXIT
0137 30141E 0138 MOV A, TXM_REG_2
013B 020139 0139 MOV P3.2, C
013D 020139 0140 MOV TXM_REG_0, A
013F 331 ; ; CHANNEL 2
0141 392B4 0142 MOV P3.2, C
0144 F555 0145 MOV TXM_REG_0, A
0147 020139 0148 JMP TXM_EXIT
0149 30141E 0150 MOV A, TXM_REG_2
014D 020139 0151 MOV P3.2, C
0151 020139 0152 MOV TXM_REG_0, A
0153 331 ; ; CHANNEL 1
0155 392B4 0156 MOV P3.2, C
0158 F555 0159 MOV TXM_REG_0, A
015B 020139 0160 JMP TXM_EXIT
015D 30141E 0161 MOV A, TXM_REG_2
0161 020139 0162 MOV P3.2, C
0163 020139 0164 MOV TXM_REG_0, A
0165 331 ; ; CHANNEL 1
0167 392B4 0168 MOV P3.2, C
016A F555 016B MOV TXM_REG_0, A
016D 020139 016C JMP TXM_EXIT

MCS-51 MACRO ASSEMBLER

LOC OBJ LINE SOURCE
0139 C1 364 TXM_EXIT: CLR C; Update compare value with
013A 742C 365 MOV A, #FULL_BIT_LOW; full bit time = 22CH
013C 75E0 366 ADD A, CCAP3L
013E F580 367 MOV CCAP3L, A
0140 742C 368 MOV A, #FULL_BIT_HIGH
0142 35FE 369 ADDC A, CCAP3H
0144 F580 370 MOV CCAP3H, A
0146 D000 371 POP PSW
0148 D000 372 POP ACC
014A 32 373 RETI

; SERIAL PORT INTERRUPT
; ================
014B C000 383 SERIAL_PORT: PUSH ACC
014C C000 384 PUSH PSW
014D 30880B 385 JNB RI, TXM
0152 E589 386 MOV A, SBUF
0154 C098 387 CLR RI
0156 3589 388 MOV SBUF, A
0158 D000 389 POP PSW
015A D000 390 POP ACC
015C 32 391 RETI

; TXM:
015D C099 392 CLR TI
015F D000 393 POP PSW
0161 D000 394 POP ACC
0163 32 395 RETI

; END

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
83C51FA/FB PCA Cookbook

BETSY JONES
ECO APPLICATIONS ENGINEER

© Intel Corporation, 1988
Order Number: 270609-001
This application note illustrates the different functions of the Programmable Counter Array (PCA) which are available on the 83C51FA and 83C51FB. Included are cookbook samples of code in typical applications to simplify the use of the PCA. Since all the examples are written in assembly language, it is assumed the reader is familiar with ASM51. For further information on these products or ASM51 refer to the Embedded Controller Handbook (Vol. I).

PCA OVERVIEW

The major new feature on the 83C51FA and 83C51FB is the Programmable Counter Array. The PCA provides more timing capabilities with less CPU intervention than the standard timer/counters. Its advantages include reduced software overhead and improved accuracy.

The PCA consists of a dedicated timer/counter which serves as the time base for an array of five compare/capture modules. Figure 1 shows a block diagram of the PCA. Notice that the PCA timer and modules are all 16-bits. If an external event is associated with a module, that function is shared with the corresponding Port 1 pin. If the module is not using the port pin, the pin can still be used for standard I/O.

Each of the five modules can be programmed in any one of the following modes:
- Rising and/or Falling Edge Capture
- Software Timer
- High Speed Output
- Watchdog Timer (Module 4 only)
- Pulse Width Modulator.

All of these modes will be discussed later in detail. However, let's first look at how to set up the PCA timer and modules.

PCA TIMER/COUNTER

The timer/counter for the PCA is a free-running 16-bit timer consisting of registers CH and CL (the high and low bytes of the count values). It is the only timer which can service the PCA. The clock input can be selected from the following four modes:
- oscillator frequency ÷ 12 (Mode 0)
- oscillator frequency ÷ 4 (Mode 1)
- Timer 0 overflows (Mode 2)
- external input on P1.2 (Mode 3)
The table below summarizes the various clock inputs for each mode at two common frequencies. In Mode 0, the clock input is simply a machine cycle count, whereas in Mode 1 the input is clocked three times faster. In Mode 2, Timer 0 overflows are counted allowing for a range of slower inputs to the timer. And finally, if the input is external the PCA timer counts 1-to-0 transitions with the maximum clock frequency equal to \( \frac{1}{6} \times \) oscillator frequency.

### Table 1. PCA Timer/Counter Inputs

<table>
<thead>
<tr>
<th>PCA Timer/Counter Mode</th>
<th>12 MHz</th>
<th>16 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode 0: fosc / 12</td>
<td>1 µsec</td>
<td>0.75 µsec</td>
</tr>
<tr>
<td>Mode 1: fosc / 4</td>
<td>330 nsec</td>
<td>250 nsec</td>
</tr>
<tr>
<td>Mode 2*: Timer 0 Overflows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timer 0 programmed in:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit mode</td>
<td>256 µsec</td>
<td>192 µsec</td>
</tr>
<tr>
<td>16-bit mode</td>
<td>65 msec</td>
<td>49 msec</td>
</tr>
<tr>
<td>8-bit auto-reload</td>
<td>1 to 255 µsec</td>
<td>0.75 to 191 µsec</td>
</tr>
<tr>
<td>Mode 3: External Input MAX</td>
<td>0.66 µsec</td>
<td>0.50 µsec</td>
</tr>
</tbody>
</table>

*In Mode 2, the overflow interrupt for Timer 0 does not need to be enabled.

Special Function Register CMOD contains the Count Pulse Select bits (CPS1 and CPS0) to specify the PCA timer input. This register also contains the ECF bit which enables an interrupt when the counter overflows. In addition, the user has the option of turning off the PCA timer during Idle Mode by setting the Counter Idle bit (CIDL). This can further reduce power consumption by an additional 30%.

**CMOD: Counter Mode Register**

<table>
<thead>
<tr>
<th>CIDL</th>
<th>WDTE</th>
<th>—</th>
<th>—</th>
<th>—</th>
<th>CPS1</th>
<th>CPS0</th>
<th>ECF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address = 0D9H</td>
<td>Reset Value = 00XX X000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not Bit Addressable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**

The user should write 0s to unimplemented bits. These bits may be used in future MCS-51 products to invoke new features, and in that case the inactive value of the new bit will be 0. When read, these bits must be treated as don’t-cares.

Table 2 lists the values for CMOD in the four possible timer modes with and without the overflow interrupt enabled. This list assumes that the PCA will be left running during Idle Mode.

### Table 2. CMOD Values

<table>
<thead>
<tr>
<th>PCA Count Pulse Selected</th>
<th>CMOD value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without interrupt enabled</td>
</tr>
<tr>
<td>Internal clock, Fosc/12</td>
<td>00 H</td>
</tr>
<tr>
<td>Internal clock, Fosc/4</td>
<td>02 H</td>
</tr>
<tr>
<td>Timer 0 overflow</td>
<td>04H</td>
</tr>
<tr>
<td>External clock at P1.2</td>
<td>06 H</td>
</tr>
</tbody>
</table>
The CCON register shown below contains the Counter Run bit (CR) which turns the timer on or off. When the PCA timer overflows, the Counter Overflow bit (CF) gets set. CCON also contains the five event flags for the PCA modules. The purpose of these flags will be discussed in the next section.

**CCON**: Counter Control Register

<table>
<thead>
<tr>
<th>CF</th>
<th>CR</th>
<th>CCF4</th>
<th>CCF3</th>
<th>CCF2</th>
<th>CCF1</th>
<th>CCF0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address = 0D8H</td>
<td>Reset Value = 00X0 0000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit Addressable

The PCA timer registers (CH and CL) can be read and written to at any time. However, to read the full 16-bit timer value simultaneously requires using one of the PCA modules in the capture mode and toggling a port pin in software. More information on reading the PCA timer is provided in the section on the Capture Mode.

**COMPARE/CAPTURE MODULES**

Each of the five compare/capture modules has a mode register called CCAPMn (n = 0, 1, 2, 3, or 4) to select which function it will perform. Note the ECCFn bit which enables an interrupt to occur when a module's event flag is set.

**CCAPMn**: Compare/Capture Mode Register

<table>
<thead>
<tr>
<th>—</th>
<th>ECOMn</th>
<th>CAPPn</th>
<th>CAPNn</th>
<th>MATn</th>
<th>TOGn</th>
<th>PWMn</th>
<th>ECCFn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address = 0DAH (n = 0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reset Value = X000 0000B</td>
</tr>
<tr>
<td>0DBH (n = 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0DCH (n = 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0DDH (n = 3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0DEH (n = 4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 lists the CCAPMn values for each different mode with and without the PCA interrupt enabled; that is, the interrupt is optional for all modes. However, some of the PCA modes require software servicing. For example, the Capture modes need an interrupt so that back-to-back events can be recognized. Also, in most applications the purpose of the Software Timer mode is to generate interrupts in software so it would be useless not to have the interrupt enabled. The PWM mode, on the other hand, does not require CPU intervention so the interrupt is normally not enabled.

**Table 3. Compare/Capture Mode Values**

<table>
<thead>
<tr>
<th>Module Function</th>
<th>CCAPMn Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without interrupt enabled</td>
</tr>
<tr>
<td>Capture Positive only</td>
<td>20H</td>
</tr>
<tr>
<td>Capture Negative only</td>
<td>10H</td>
</tr>
<tr>
<td>Capture Pos. or Neg.</td>
<td>30H</td>
</tr>
<tr>
<td>Software Timer</td>
<td>48H</td>
</tr>
<tr>
<td>High Speed Output</td>
<td>4CH</td>
</tr>
<tr>
<td>Watchdog Timer</td>
<td>48 or 4CH</td>
</tr>
<tr>
<td>Pulse Width Modulator</td>
<td>42H</td>
</tr>
</tbody>
</table>
It should be mentioned that a particular module can change modes within the program. For example, a module might be used to sample incoming data. Initially it could be set up to capture a falling edge transition. Then the same module can be reconfigured as a software timer to interrupt the CPU at regular intervals and sample the pin.

Each module also has a pair of 8-bit compare/capture registers (CCAPnH, CCAPnL) associated with it. These registers are used to store the time when a capture event occurred or when a compare event should occur. Remember, event times are based on the free-running PCA timer (CH and CL). For the PWM mode, the high byte register CCAPnH controls the duty cycle of the waveform.

When an event occurs, a flag in CCON is set for the appropriate module. This register is bit addressable so that event flags can be checked individually.

**CCON: Counter Control Register**

<table>
<thead>
<tr>
<th>CF</th>
<th>CR</th>
<th>—</th>
<th>CCF4</th>
<th>CCF3</th>
<th>CCF2</th>
<th>CCF1</th>
<th>CCF0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address = 008H</td>
<td>Reset Value = 0000 0000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Addressable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These five event flags plus the PCA timer overflow flag share an interrupt vector as shown below. These flags are not cleared when the hardware vectors to the PCA interrupt address (0033H) so that the user can determine which event caused the interrupt. This also allows the user to define the priority of servicing each module.

![Figure 2. PCA Interrupt](image)

An additional bit was added to the Interrupt Enable (IE) register for the PCA interrupt. Similarly, a high priority bit was added to the Interrupt Priority (IP) register.

**IE: Interrupt Enable Register**

<table>
<thead>
<tr>
<th>EA</th>
<th>EC</th>
<th>ET2</th>
<th>ES</th>
<th>ET1</th>
<th>EX1</th>
<th>ET0</th>
<th>EX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address = 0A8H</td>
<td>Reset Value = 0000 0000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Addressable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IP: Interrupt Priority Register**

<table>
<thead>
<tr>
<th>—</th>
<th>PPC</th>
<th>PT2</th>
<th>PS</th>
<th>PT1</th>
<th>PX1</th>
<th>PT0</th>
<th>PX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address = 0B8H</td>
<td>Reset Value = X000 0000B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Addressable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember, each of the six possible sources for the PCA interrupt must be individually enabled as well—in the CCAPMn register for the modules and in the CCON register for the timer.
**CAPTURE MODE**

Both positive and negative transitions can trigger a capture with the PCA. This allows the PCA flexibility to measure periods, pulse widths, duty cycles, and phase differences on up to five separate inputs. This section gives examples of all these different applications.

Figure 3 shows how the PCA handles a capture event. Using Module 0 for this example, the signal is input to P1.3. When a transition is detected on that pin, the 16-bit value of the PCA timer (CH,CL) is loaded into the capture registers (CCAP0H,CCAP0L). Module 0's event flag is set and an interrupt is flagged. The interrupt will then be generated if it has been properly enabled.

In the interrupt service routine, the 16-bit capture value must be saved in RAM before the next event capture occurs; a subsequent capture will write over the first capture value. Also, since the hardware does not clear the event flag, it must be cleared in software.

The time it takes to service this interrupt routine determines the resolution of back-to-back events with the same PCA module. To store two 8-bit registers and clear the event flag takes at least 9 machine cycles. That includes the call to the interrupt routine. At 12 MHz, this routine would take less than 10 microseconds. However, depending on the frequency and interrupt latency, the resolution will vary with each application.

**Measuring Pulse Widths**

To measure the pulse width of a signal, the PCA module must capture both rising and falling edges (see Figure 4). The module can be programmed to capture either edge if it is known which edge will occur first. However, if this is not known, the user can select which edge will trigger the first capture by choosing the proper mode for the module.

Listing 1 shows an example of measuring pulse widths. (It's assumed the incoming signal matches the one in Figure 4.) In the interrupt routine the first set of capture values are stored in RAM. After the second capture, a subtraction routine calculates the pulse width in units of PCA timer ticks. Note that the subtraction does not have to be completed in the interrupt service routine. Also, this example assumes that the two capture events will occur within 216 counts of the PCA timer, i.e. rollovers of the PCA timer are not counted.

![Figure 3. PCA Capture Mode (Module 0)](image)

![Figure 4. Measuring Pulse Width](image)

---

2-249
### Listing 1. Measuring Pulse Widths

<table>
<thead>
<tr>
<th>RAM locations to store capture values</th>
<th>DATA</th>
<th>3OH</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPTURE</td>
<td>DATA</td>
<td>32H</td>
</tr>
<tr>
<td>PULSE_WIDTH</td>
<td>DATA</td>
<td>20H</td>
</tr>
<tr>
<td>FLAG</td>
<td>BIT</td>
<td>0</td>
</tr>
</tbody>
</table>

ORG OOOOH
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT

PCA_INIT:

- Initialize PCA timer
- Input to timer = 1/12 x Fosc
- Initialize Module 0 in capture mode
- Capture positive edge first
- for measuring pulse width

MOV CMOD, #00H
MOV CH, #00H
MOV CL, #00H

; Enable PCA interrupt
SETB EC
SETB EA
SETB CR
CLR FLAG

; Clear Module 0's event flag
; Check if this is the first capture or second
FIRST_CAPTURE:

MOV CAPTURE, CCAPOL
MOV CAPTURE+1, CCAPOH
MOV CCAPMO, #11H
SETB FLAG

; Save 16-bit capture value
; in RAM
; Change module to now capture
; falling edges
; Signify 1st capture complete
RETI

SECOND_CAPTURE:

PUSH ACC
PUSH PSW
CLR C
MOV A, CCAPOL
SUBB A, CAPTURE
MOV PULSE_WIDTH, A
MOV A, CCAPOH
SUBB A, CAPTURE+1
MOV PULSE_WIDTH+1, A

; Optional--needed if user wants to
; measure next pulse width
MOV CCAPMO, #21H
CLR FLAG
POP PSW
POP ACC
RETI
Measuring Periods

Measuring the period of a signal with the PCA is similar to measuring the pulse width. The only difference will be the trigger source for the capture mode. In Figure 5, rising edges are captured to calculate the period. The code is identical to Listing 1 except that the capture mode should not be changed in the interrupt routine. The result of the subtraction will be the period.

\[
\text{Time (Capture 2) - Time (Capture 1) = Period}
\]

Figure 5. Measuring Period

Measuring Frequencies

Measuring a frequency with the PCA capture mode involves calculating a sample time for a known number of samples. In Figure 6, the time between the first capture and the “Nth” capture equals the sample time T. Listing 2 shows the code for \( N = 10 \) samples. It’s assumed that the sample time is less than \( 2^{16} \) counts of the PCA timer.

\[
\text{Frequency} = \frac{N}{T} = \frac{\# \text{ of Samples}}{\text{Sample Time}}
\]

Figure 6. Measuring Frequency
## Listing 2. Measuring Frequencies

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>30H</td>
<td>CAPTURE</td>
</tr>
<tr>
<td>32H</td>
<td>PERIOD</td>
</tr>
<tr>
<td>34H</td>
<td>SAMPLE_COUNT</td>
</tr>
<tr>
<td>20H.0</td>
<td>FLAG</td>
</tr>
</tbody>
</table>

; RAM locations to store capture values

ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT

; PCA_INIT:
; Initialization of PCA timer, Module 0, and interrupt is the
; same as in Listing 1. Also need to initialize the sample
; count.

MOV SAMPLE_COUNT, #10D ; N = 10 for this example

; Main program goes here

; This code assumes only Module 0 is being used.

PCA_INTERRUPT:
    CLR CCFO ; Clear module 0's event flag
    JB FLAG, NEXT_CAPTURE

FIRST_CAPTURE:
    MOV CAPTURE, CCAPOL
    MOV CAPTURE+1, CCAPOH
    SETB FLAG ; Signify first capture complete
    RETI

NEXT_CAPTURE:
    DJNZ SAMPLE_COUNT, EXIT
    PUSH ACC
    PUSH PSW
    CLR C
    MOV A, CCAPOL ; 16-bit subtraction
    SUBB A, CAPTURE
    MOV PERIOD, A
    MOV A, CCAPOH
    SUBB A, CAPTURE+1
    MOV PERIOD+1, A
    MOV SAMPLE_COUNT, #10D ; Reload for next period
    CLR FLAG
    POP PSW
    POP ACC
EXIT:
    RETI
The user may instead want to measure frequency by counting pulses for a known sample time. In this case, one module is programmed in the capture mode to count edges (either rising or falling), and a second module is programmed as a software timer to mark the sample time. An example of a software timer is given later. For information on resolution in measuring frequencies, refer to Article Reprint AR-517, “Using the 8051 Microcontroller with Resonant Transducers,” in the Embedded Controller Handbook.

**Measuring Duty Cycles**

To measure the duty cycle of an incoming signal, both rising and falling edges need to be captured. Then the duty cycle must be calculated based on three capture values as seen in Figure 7. The same initialization routine is used from the previous example. Only the PCA interrupt service routine is given in Listing 3.

### Figure 7. Measuring Duty Cycle

**Listing 3. Measuring Duty Cycle**

```assembly
; RAM locations to store capture values
  CAPTURE   DATA  30H
  PULSE_WIDTH DATA  32H
  PERIOD    DATA  34H
  FLAG_1    BIT   20H.0
  FLAG_2    BIT   20H.1

; ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT

; PCA_INIT:
; Initialization for PCA timer, module, and interrupt the same
; as in Listing 1. Capture positive edge first, then either
; edge.
; ****************************
; Main program goes here
; ****************************

; This code assumes only Module 0 is being used.
PCA_INTERRUPT:
  CLR CCF0 ; Clear Module 0's event flag
  JB FLAG_1, SECOND_CAPTURE

FIRST_CAPTURE:
  MOV CAPTURE, CCAPOL
  MOV CAPTURE+1, CCAPOLH
  SETB FLAG_1  ; Signify first capture complete
  MOV CCAPMO, #31H  ; Capture either edge now
  RETI
```

2-253
Listing 3. Measuring Duty Cycle (Continued)

; SECOND_CAPTURE:
PUSH ACC
PUSH PSW
JB FLAG_2, THIRD_CAPTURE
CLR C ; Calculate pulse width
MOV A, CCAPOL ; 16-bit subtract
SUBB A, CAPTURE
MOV PULSE_WIDTH, A
MOV A, CCAPOH
SUBB A, CAPTURE+1
MOV PULSE_WIDTH+1, A
SETB FLAG_2 ; Signify second capture complete
POP PSW
POP ACC
RETI

; THIRD_CAPTURE:
CLR C ; Calculate period
MOV A, CCAPOL ; 16-bit subtract
SUBB A, CAPTURE
MOV PERIOD, A
MOV A, CCAPOH
SUBB A, CAPTURE+1
MOV PERIOD+1, A
MOV CCAPMO, #21H ; Optional - reconfigure module to
capture positive edges for next
CLR FLAG_1
CLR FLAG_2 ; cycle
POP PSW
POP ACC
RETI

After the third capture, a 16-bit by 16-bit divide routine needs to be executed. This routine is located in Appendix B. Due to its length, it's up to the user whether the divide routine should be completed in the interrupt routine or be called as a subroutine from the main program.

Measuring Phase Differences

Because the PCA modules share the same time base, the PCA is useful for measuring the phase difference between two or more signals. For this example, two signals are input to Modules 0 and 1 as seen in Figure 8. Both modules are programmed to capture rising edges only. Listing 4 shows the code needed to measure the difference between these two signals. This code does not assume one signal is leading or lagging the other.

![Figure 8. Measuring Phase Differences](image-url)
Listing 4. Measuring Phase Differences

; RAM locations to store capture values
CAPTURE_O DATA 30H
CAPTURE_1 DATA 32H
PHASE DATA 34H
FLAG_O BIT 20H.0
FLAG_1 BIT 20H.1
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT
;
PCA_INIT:
; Same initialization for PCA timer, and interrupt as
; in Listing 1. Initialize two PCA modules as follows:
;
MOV CCAPMO, #21H ; Module 0 capture rising edges
MOV CCAPM1, #21H ; Module 1 same

; *******************************************************
; Main program goes here
; *******************************************************
;
PCA_INTERRUPT:
JB CCF0, MODULE_0 ; Determine which module's
JB CCF1, MODULE_1 ; event caused the interrupt
;
MODULE_0:
CLR CCF0 ; Clear Module 0's event flag
MOV CAPTURE_0, CCAPOL ; Save 16-bit capture value
MOV CAPTURE_0+1, CCAPOLH
JB FLAG_1, CALCULATE_PHASE ; If capture complete on
; Module 1, go to calculation
SETB FLAG_0 ; Signify capture on Module 0
RETI
Listing 4. Measuring Phase Differences (Continued)

```
MODULE_1:
  CLR CCF_1 ; Clear Module 1's event flag
  MOV CAPTURE_1, CCAP1L
  MOV CAPTURE_1+1, CCAP1H
  JB FLAG_O, CALCULATE_PHASE ; If capture complete on
  MOV CAPTURE_1, CCAP1L ; Module 0, go to calculation
  MOV CAPTURE_1+1, CCAP1H ; Signify capture on Module 1
  SETB FLAG_1
  RETI

CALCULATE_PHASE:
  PUSH ACC
  PUSH PSW
  CLR C
  JB FLAG_O, MODO_LEADING ; This calculation does not
  JB FLAG_1, MODI LEADING ; have to be completed in the
                          ; interrupt service routine

MODO_LEADING:
  MOV A, CAPTURE_1
  SUBB A, CAPTURE_0
  MOV PHASE, A
  MOV A, CAPTURE_1+1
  SUBB A, CAPTURE_0+1
  MOV PHASE+1, A
  CLR FLAG_O
  JMP EXIT

MODI LEADING:
  MOV A, CAPTURE_0
  SUBB A, CAPTURE_1
  MOV PHASE, A
  MOV A, CAPTURE_0+1
  SUBB A, CAPTURE_1+1
  MOV PHASE+1, A
  CLR FLAG_1
EXIT:
  POP PSW
  POP ACC
  RETI
```
Reading the PCA Timer

Some applications may require that the PCA timer be read instantaneously as a real-time event. Since the timer consists of two 8-bit registers (CH,CL), it would normally take two MOV instructions to read the whole timer. An invalid read could occur if the registers rolled over in the middle of the two MOVs.

However, with the capture mode a 16-bit timer value can be loaded into the capture registers by toggling a port pin. For example, configure Module 0 to capture falling edges and initialize P1.3 to be high. Then when the user wants to read the PCA timer, clear P1.3 and the full 16-bit timer value will be saved in the capture registers. It's still optional whether the user wants to generate an interrupt with the capture.

COMPARE MODE

In this mode, the 16-bit value of the PCA timer is compared with a 16-bit value pre-loaded in the module's compare registers. The comparison occurs three times per machine cycle in order to recognize the fastest possible clock input, i.e. ¼ x oscillator frequency. When there is a match, one of three events can happen:

1. an interrupt — Software Timer mode
2. toggle of a port pin — High Speed Output mode
3. a reset — Watchdog Timer mode.

Examples of each compare mode will follow.

SOFTWARE TIMER

In most applications a software timer is used to trigger interrupt routines which must occur at periodic intervals. Figure 9 shows the sequence of events for the Software Timer mode. The user preloads a 16-bit value in a module's compare registers. When a match occurs between this compare value and the PCA timer, an event flag is set and an interrupt is flagged. An interrupt is then generated if it has been enabled.

If necessary, a new 16-bit compare value can be loaded into (CCAP0H, CCAP0L) during the interrupt routine. The user should be aware that the hardware temporarily disables the comparator function while these registers are being updated so that an invalid match will not occur. That is, a write to the low byte (CCAPn0) disables the comparator while a write to the high byte (CCAPnH) re-enables the comparator. For this reason, user software must write to CCAP0L first, then CCAP0H. The user may also want to hold off any interrupts from occurring while these registers are being updated. This can easily be done by clearing the EA bit. See the code example in Listing 5.

![Figure 9. Software Timer Mode (Module 0)](270609-9)
Listing 5. Software Timer

; Generate an interrupt in software every 20 msec
;
; Frequency = 12 MHz
; PCA clock input = 1/12 x Fosc → 1 μsec
;
; Calculate reload value for compare registers:
; 20 msec
; ----------------- = 20,000 counts
; 1 μsec/count
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT

; PCA_INIT:
; Initialize PCA timer same as in Listing 1
; MOV CCAPO, #49H ; Module 0 in Software Timer mode
MOV CCAPOL, #LOW(20000) ; Write to low byte first
MOV CCAPOH, #HIGH(20000)

SETB EC ; Enable PCA interrupt
SETB EA
SETB CR ; Turn on PCA timer

; ***************************************************************************
; Main program goes here
; ***************************************************************************

; PCA_INTERRUPT:
CLR CCFO ; Clear Module 0's event flag
PUSH ACC
PUSH PSW
CLR EA ; Hold off interrupts
MOV A, #LOW(20000) ; 16-Bit Add
ADD A, CCAPOL ; Next match will occur
MOV CCAPOL, A
MOV A, #HIGH(20000)
ADDC A, CCAPOH
MOV CCAPOH, A
SETB EA

; Continue with routine
;
POP PSW
POP ACC
RETI
HIGH SPEED OUTPUT

The High Speed Output (HSO) mode toggles a port pin when a match occurs between the PCA timer and the pre-loaded value in the compare registers (see Figure 10). The HSO mode is more accurate than toggling pins in software because the toggle occurs before branching to an interrupt, i.e. interrupt latency will not affect the accuracy of the output. In fact, the interrupt is optional. Only if the user wants to change the time for the next toggle is it necessary to update the compare registers. Otherwise, the next toggle will occur when the PCA timer rolls over and matches the last compare value. Examples of both are shown.

![Diagram of High Speed Output Mode](270609-10)

Without any CPU intervention, the fastest waveform the PCA can generate with the HSO mode is a 30.5 Hz signal at 16 MHz. Refer to Listing 6. By changing the PCA clock input, slower waveforms can also be generated.

**Listing 6. High Speed Output (Without Interrupt)**

```
; Maximum output with HSO mode without interrupts = 30.5 Hz signal
; Frequency = 16 MHz
; PCA clock input = 1/4 x Fosc → 250 nsec

MOV CMOD, #02H
MOV CL, #00H
MOV CH, #00H
MOV CCAPMO, #4CH ; HSO mode without interrupt enabled
MOV CCAPOL, #0FFH ; Write to low byte first
MOV CCAPOH, #0FFH ; P1.3 will toggle every 2^16 counts
                  ; or 16.4 msec
                  ; Period = 30.5 Hz
SETB CR           ; Turn on PCA timer
```

2-259
In this next example, the PCA interrupt is used to change the compare value for each toggle. This way a variable frequency output can be generated. Listing 7 shows an output of 1 KHz at 16 Mhz.

Listing 7. High Speed Output (With Interrupt)

```assembly
ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

; PCA_INIT:
MOV CMOD, #02H ; Clock input = 250 nsec
MOV CL, #00H ; at 16 MHz
MOV CH, #00H
MOV CCAPMO, #4DH ; Module 0 in HSO mode with
MOV CCAFOL, #LOW(1000) ; PCA interrupt enabled
MOV CCAFOL, #HIGH(1000) ; t = 1000 (arbitrary)
CLR Pl.3

; SETB EC ; Enable PCA interrupt
SETB EA ; Turn on PCA timer
SETB CR

; ****************************************
; Main program goes here
; ****************************************

; This code assumes only Module 0 is being used.

PCA_INTERRUPT:
CLR CCFO ; Clear Module 0's event flag
PUSH ACC
PUSH PSW
CLR EA
MOV A, #LOW(2000) ; Hold off interrupts
ADD A, CCAFOL ; 16-bit add
MOV CCAFOL, A ; 2000 counts later, Pl.3
MOV A, #HIGH(2000) ; will toggle
ADDC A, CCAFOL
MOV CCAFOL, A
SETB EA
POP PSW
POP ACC
RETI
```

---

500 µsec

\[
\frac{500 \, \text{µsec}}{250 \, \text{nsec/count}} = 2000 \, \text{counts}
\]

270609-11
Another option with the HSO mode is to generate a single pulse. Listing 8 shows the code for an output with a pulse width of 20 µsec. As in the previous example, the PCA interrupt will be used to change the time for the toggle. The first toggle will occur at time “t”. After 80 counts of the PCA timer, 20 µsec will have expired, and the next toggle will occur. Then the HSO mode will be disabled.

**Listing 8. High Speed Output (Single Pulse)**

```
ORG 0000H
JMP PCA.INIT
ORG 0033H
JMP PCA.INTERRUPT

; PCA.INIT:
MOV CMOD, #02H ; Clock input = 250 nsec
MOV CL, #00H ; at 16 MHz
MOV CH, #00H
MOV CCAPMO, #4DH ; Module 0 in HSO mode with
MOV CCAPOL, #LOW(1000) ; PCA interrupt enabled
MOV CCAPOH, #HIGH(1000) ; t = 1000 (arbitrary)
CLR Pl.3
;
SETH EC ; Enable PCA interrupt
SETH EA
SETB CR ; Turn on PCA timer
;
; *******************************************************
; Main program goes here
; *******************************************************
;
; This code assumes only Module 0 is being used.
PCA.INTERRUPT:

CLR CCFO ; Clear Module 0’s event flag
JNB Pl.3, DONE
;
PUSH ACC
PUSH PSW
CLR EA ; Hold off interrupts
MOV A, #LOW(80) ; 16-bit add
ADD A, CCAPOL
MOV CCAPOL, A
MOV A, #HIGH(80)
ADDCA A, CCAPOH
MOV CCAPOH, A
SETH EA
POP PSW
POP ACC
RETI
;
DONE:
MOV CCAPMO, #00H ; Disable HSO mode
RETI
```

\[ \frac{20 \mu \text{sec}}{250 \text{ nsec/count}} = 80 \text{ counts} \]
WATCHDOG TIMER

An on-board watchdog timer is available with the PCA to improve the reliability of the system without increasing chip count. Watchdog timers are useful for systems which are susceptible to noise, power glitches, or electrostatic discharge. Module 4 is the only PCA module which can be programmed as a watchdog. However, this module can still be used for other modes if the watchdog is not needed.

Figure 11 shows a diagram of how the watchdog works. The user pre-loads a 16-bit value in the compare registers. Just like the other compare modes, this 16-bit value is compared to the PCA timer value. If a match is allowed to occur, an internal reset will be generated. This will not cause the RST pin to be driven high.

In order to hold off the reset, the user has three options:

1. periodically change the compare value so it will never match the PCA timer,
2. periodically change the PCA timer value so it will never match the compare value, or
3. disable the watchdog by clearing the WDTE bit before a match occurs and then re-enable it.

The first two options are more reliable because the watchdog timer is never disabled as in option #3. If the program counter ever goes astray, a match will eventually occur and cause an internal reset. The second option is also not recommended if other PCA modules are being used. Remember, the PCA timer is the time base for all modules; changing the time base for other modules would not be a good idea. Thus, in most applications the first solution is the best option.

Listing 9 shows the code for initializing the watchdog timer. Module 4 can be configured in either compare mode, and the WDTE bit in CMOD must also be set. The user’s software then must periodically change (CCAP4H,CCAP4L) to keep a match from occurring with the PCA timer (CH,CL). This code is given in the WATCHDOG routine.

This routine should not be part of an interrupt service routine. Why? Because if the program counter goes astray and gets stuck in an infinite loop, interrupts will still be serviced and the watchdog will keep getting reset. Thus, the purpose of the watchdog would be defeated. Instead call this subroutine from the main program within 2^16 count of the PCA timer.

Figure 11. Watchdog Timer Mode (Module 4)
Listing 9. Watchdog Timer

INIT_WATCHDOG:
    MOV CCAPM4, #4CH ; Module 4 in compare mode
    MOV CCAP4L, #OFFH ; Write to low byte first
    MOV CCAP4H, #OFFH ; Before PCA timer counts up to
    ORL CMOD, #40H ; FFFF Hex, these compare values
                     ; must be changed
                     ; Set the WDTE bit to enable the
                     ; watchdog timer without changing
                     ; the other bits in CMOD

; Main program goes here, but CALL WATCHDOG periodically.

WATCHDOG:
    CLR EA ; Hold off interrupts
    MOV CCAP4L, #00 ; Next compare value is within
    MOV CCAP4H, CH ; 255 counts of the current PCA
    SETB EA ; timer value
    RET

PULSE WIDTH MODULATOR

The PCA can generate 8-bit PWMs by comparing the low byte of the PCA timer (CL) with the low byte of the compare registers (CCAPnL). When CL < CCAPnL the output is low. When CL ≥ CCAPnL the output is high.

To control the duty cycle of the output, the user actually loads a value into the high byte CCAPnH (see Figure 12). Since a write to this register is asynchronous, a new value is not shifted into CCAPnL for comparison until the next period of the output: that is, when CL rolls over from 255 to 00. This mechanism provides “glitch-free” writes to CCAPnH when the duty cycle of the output is changed.

CCAPnH can contain any integer from 0 to 255, but Figure 13 shows a few common duty cycles and the corresponding values for CCAPnH. Note that a 0% duty cycle can be obtained by writing to the port pin directly with the CLR bit instruction. To calculate the CCAPnH value for a given duty cycle, use the following equation:

\[
CCAPnH = 256 (1 - \text{Duty Cycle})
\]

where CCAPnH is an 8-bit integer and Duty Cycle is expressed as a fraction.
Figure 13. CCAPnH Varies Duty Cycle

Table 4. PWM Frequencies.

<table>
<thead>
<tr>
<th>PCA Timer Mode</th>
<th>PWM Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 MHz</td>
</tr>
<tr>
<td>1/12 Osc. Frequency</td>
<td>3.9 KHz</td>
</tr>
<tr>
<td>1/4 Osc. Frequency</td>
<td>11.8 KHz</td>
</tr>
<tr>
<td>Timer 0 Overflow:</td>
<td></td>
</tr>
<tr>
<td>8-bit</td>
<td>15.5 Hz</td>
</tr>
<tr>
<td>16-bit</td>
<td>0.06 Hz</td>
</tr>
<tr>
<td>8-bit Auto-Reload</td>
<td>3.9 KHz to 15.3 Hz</td>
</tr>
<tr>
<td>External Input (Max)</td>
<td>5.9 KHz</td>
</tr>
</tbody>
</table>
Listing 10. PWM

```
INIT-PWM:
    MOV CMOD, #02H ; Clock input = 250 nsec at 16 MHz
    MOV CL, #00H ; Frequency of output = 15.6 KHz
    MOV CH, #00H
    MOV CCAPMO, #42H ; Module 0 in PWM mode
    MOV CCAPOL, #00H
    MOV CCAPOH, #128D ; 50 percent duty cycle

    ; SETB CR ; Turn on PCA timer
```

The frequency of the PWM output will depend on which of the four inputs is chosen for the PCA timer. The maximum frequency is 15.6 KHz at 16 MHz. Refer to Table 4 for a summary of the different PWM frequencies possible with the PCA.

Listing 10 shows how to initialize Module 0 for a PWM signal at 50% duty cycle. Notice that no PCA interrupt is needed to generate the PWM (i.e. no software overhead). To create a PWM output on the 8051 requires a hardware timer plus software overhead to toggle the port pin. The advantage of the PCA is obvious, not to mention it can support up to 5 PWM outputs with just one chip.

**CONCLUSION**

This list of examples with the PCA is by no means exhaustive. However, the advantages of the PCA can easily be seen from the given applications. For example, the PCA can provide better resolution than Timers 0, 1 and 2 because the PCA clock rate can be three times faster. The PCA can also perform many tasks that these hardware timers can not, i.e. measure phase differences between signals or generate PWMs. In a sense, the PCA provides the user with five more timer/counters in addition to Timers 0, 1 and 2 on the 8XC51FA/FB.

Appendix A includes test routines for all the software examples in this application note. The divide routine for calculating duty cycles is in Appendix B. And finally, Appendix C is a table of the Special Function Registers for the 8XC51FA/FB with the new or modified registers **boldfaced**.
APPENDIX A
TEST ROUTINES

Listing 1a - Measuring Pulse Widths

$nomod51
$symbols
$nolist
$include (reg252.pdf)
$list

Variables

CAPTURE DATA 30H
PULSE_WIDTH DATA 32H
FLAG BIT 20H.0

ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

: Initialize PCA timer
PCA_INIT: MOV CMOD, #00H
MOV CH, #00
MOV CL, #00
: Initialize Module 0 in capture mode
MOV CCAPM0, #21H
MOV CCAP0H, #00
MOV CCAP0L, #00
: SETB EC
SETB EA
SETB CR
CLR FLAG

; Input to PCA timer = 1/12 x Fosc
; Capture positive edge first on P1.3
; Enable PCA interrupt
; Turn PCA timer on
; Clear test flag

; Test program only
WAIT: JMP $; Wait for PCA interrupt
JMP WAIT

; This code assumes Module 0 is the only module being used. If
; other PCA module's are being used, software must check which
; module's event flag caused the interrupt.

PCJ_INTERRUPT:
CLR CCF0
JB FLAG, SECOND_CAPTURE

FIRST_CAPTURE:
MOV CAPTURE, CCAP0L
MOV CAPTURE+1, CCAP0H

270609-16
MOV CCAPM0, #11H ; Change module to now capture
SETB FLAG ; falling edges
RETI ; Signify first capture complete

SECOND_CAPTURE:
  PUSH ACC
  PUSH PSW
  CLR C
  MOV A, CCAPOL
  SUBB A, CAPTURE
  MOV PULSE_WIDTH, A
  MOV A, CCAPOH
  SUBB A, CAPTURE+1
  MOV PULSE_WIDTH+1, A

MOV CCAPM0, #21H ; Optional if user wants to measure
  CLR FLAG ; next pulse width
  POP PSW
  POP ACC
  RETI

END
Listing 1b - Measuring Periods

; nomod51
; nosymbols
; noist
; include (reg252.pdf)
; list

; Variables

CAPTURE DATA 30H
PERIOD DATA 32H
FLAG BIT 20H.0

ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

; Initialize PCA timer
PCA_INIT: MOV CMOD, #00H ; Input to timer = 1/12 x Fosc
MOV CH, #00H
MOV CL, #00

; Initialize Module 0 in capture mode
MOV CCAPM0, #21H ; Capture rising edges on P1.3
MOV CCAPOL, #00
MOV CCAPOH, #00

SETB EC ; Enable PCA interrupt
SETB EA
SETB CR ; Turn PCA timer on
CLR FLAG ; Clear test flag

; Test program only

WAIT: JMP $ ; Wait for PCA interrupt
JMP WAIT

; This code assumes only Module 0 is being used. If other modules
; are being used, software must check which module’s flag caused
; the interrupt.

PCA_INTERRUPT:
CLR CCF0 ; Clear module 0’s event flag
JB FLAG, SECOND_CAPTURE

FIRST_CAPTURE:
MOV CAPTURE, CCAPOL
MOV CAPTURE+1, CCAP0H
SETB FLAG
RETI

; Signify first capture complete

SECOND_CAPTURE:
PUSH ACC
PUSH PSW
CLR C
MOV A, CCAP0L
SUBB A, CAPTURE
MOV PERIOD, A
MOV A, CCAP0H
SUBB A, CAPTURE+1
MOV PERIOD+1, A

; 16-Bit subtraction

; CLR FLAG
POP PSW
POP ACC
RETI

END
Listing 2 - Measuring Frequencies

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPTURE</td>
<td>DATA</td>
<td>30H</td>
</tr>
<tr>
<td>PERIOD</td>
<td>DATA</td>
<td>32H</td>
</tr>
<tr>
<td>SAMPLE_COUNT</td>
<td>DATA</td>
<td>34H</td>
</tr>
<tr>
<td>FLAG</td>
<td>BIT</td>
<td>20H.0</td>
</tr>
</tbody>
</table>

ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

Initialize PCA timer

PCA_INIT:
MOV CMOD, #00H
MOV CH, #00
MOV CL, #00

Initialize Module 0 in capture mode

MOV CCAPM0, #21H
MOV CCAP0H, #00
MOV CCAP0L, #00

MOV SAMPLE_COUNT, #10D

SETB EC
SETB EA
SETB CR
CLR FLAG

; Input to PCA timer = 1/12 x Fosc

Test program only

WAIT:
JMP $ ; Wait for PCA interrupt
JMP WAIT

This code assumes only Module 0 is being used.

PCA_INTERRUPT:
CLR CCF0 ; Clear module 0's event flag

JB FLAG, NEXT_CAPTURE

270609–20
FIRST_CAPTURE:
    MOV CAPTURE, CCAP0L
    MOV CAPTURE+1, CCAP0H
    SETB FLAG
    RETI
    ; Signify first capture complete

NEXT_CAPTURE:
    DJNZ SAMPLE_COUNT, EXIT
    PUSH ACC
    PUSH PSW
    CLR C
    MOV A, CCAP0L
    SUBB A, CAPTURE
    MOV PERIOD, A
    MOV A, CCAP0H
    SUBB A, CAPTURE+1
    MOV PERIOD+1, A
    ; 16-Bit subtraction
    MOV SAMPLE_COUNT, #10D
    CLR FLAG
    POP PSW
    POP ACC
    EXIT:
    RETI
    ; Reload for next capture

END
List 3 - Measuring Duty Cycle

Variables

| CAPTURE   | DATA  | 30H |
| PULSE_WIDTH | DATA  | 32H |
| PERIOD    | DATA  | 34H |

FLAG_1 BIT 20H.0
FLAG_2 BIT 20H.1

ORG 0000H
JMP PCA_INIT

ORG 0023H
JMP PCA_INTERRUPT

Initialize PCA timer
PCA_INIT: MOV CMOD, #00H ; Input to PCA timer = 1/12 x Fosc
MOV CH, #00
MOV CL, #00

Initialize Module 0 in capture mode
MOV CCAPM0, #21H ; Capture positive edge first on P1.3
MOV CCAP0H, #00
MOV CCAP0L, #00

CLR FLAG_1
CLR FLAG_2 ; Clear test flags

SETB EC
SETB EA ; Enable PCA interrupt
SETB CR ; Turn PCA timer on

......

Test program only

WAIT: JMP $ ; Wait for PCA interrupt
JMP WAIT

.....

This code assumes Module 0 is the only PCA module being used.

PCA_INTERRUPT:
CLR CCF0 ; Clear module 0's event flag
JB FLAG_1, SECOND_CAPTURE

270609-22
FIRST_CAPTURE:
MOV CAPTURE, CCAP0L
MOV CAPTURE+1, CCAP0H
SETB FLAG_1 ; Signify first capture complete
MOV CCAPM0, #31H ; Capture either edge now
RETI

SECOND_CAPTURE:
PUSH ACC
PUSH PSW
JB FLAG_2, THIRD_CAPTURE
CLR C
MOV A, CCAP0L
SUBB A, CAPTURE
MOV PULSE_WIDTH, A
MOV A, CCAP0H
SUBB A, CAPTURE+1
MOV PULSE_WIDTH+1, A
SETB FLAG_2 ; Signify second capture complete
POP PSW
POP ACC
RETI

THIRD_CAPTURE:
CLR C ; Calculate period
MOV A, CCAP0L
SUBB A, CAPTURE
MOV PERIOD, A
MOV A, CCAP0H
SUBB A, CAPTURE+1
MOV PERIOD+1, A
MOV CCAPM0, #21H ; Optional- reconfigure module to capture positive edges for next cycle
CLR FLAG_1
CLR FLAG_2
POP PSW
POP ACC
RETI

END
Listing 4 - Measuring Phase Differences

$nomod51
$symbols
$nolist
$include (reg252.pdf)
$list

Variables

CAPTURE_0       DATA       0H
CAPTURE_1       DATA       2H
PHASE           DATA       3H
FLAG_0          BIT        0H.0
FLAG_1          BIT        0H.1

ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

; Initialize PCA timer
; Input to PCA timer = 1/12 x Fosc
PCA_INIT:  MOV CMOD, #00H
           MOV CH, #00
           MOV CL, #00

; Initialize Modules 0 & 1 in capture mode
; Capture positive edges on P1.3
           MOV CCAPMO, #21H
           MOV CCAP0H, #00
           MOV CCAPOL, #00

           MOV CCAPM1, #21H
           MOV CCAP1H, #00
           MOV CCAP1L, #00

           MOV RO, #0FFH
           MOV R1, #0FFH

; Used for test program only
; Clear test flags
           CLR FLAG_0
           CLR FLAG_1

; Enable PCA interrupt
           SETB EC
           SETB EA
           SETB CR

; Turn PCA timer on

270609-24
**Test program only**

```
; Generate two waveforms
MAIN:
CALL TOG1                ; with known phase difference
CALL TOG2
JMP MAIN

; These two waveforms are input to
TOG1:
CPL P1.6
CALL DELAY1
RET

TOG2:
CPL P1.5
CALL DELAY1
RET

; DELAY1:
DJNZ R0, $        ; P1.3 and P1.4
RET

; DELAY2:
DJNZ R1, $        
RET
```

---

This code assumes only Modules 0 and 1 are being used.

**PCA_INTERRUPT:**

```
JB CCF0, MODULE_0    ; Determine which module's event
JB CCF1, MODULE_1    ; caused the interrupt

; Clear Module 0's event flag
MODULE_0:
CLR CCF0
MOV CAPTURE_0, CCAP0L
MOV CAPTURE_0+1, CCAP0H
JB FLAG_0, CALCULATE_PHASE
SETB FLAG_0        ; Signify capture complete on
RETI                ; Module 0

; Clear Module 1's event flag
MODULE_1:
CLR CCF1
MOV CAPTURE_1, CCAP1L
MOV CAPTURE_1+1, CCAP1H
JB FLAG_0, CALCULATE_PHASE
SETB FLAG_1        ; Signify capture complete
RETI                ; Module 1

; This calculation does not have to
CALCULATE_PHASE:
PUSH ACC            ; be completed in the Interrupt
PUSH PSW
CLR C               ; service routine

; JB FLAG_0, MOD0_LEADING
```

270609-25
JB FLAG_1, MOD1_LEADING

MOD0_LEADING:
  MOV A, CAPTURE_1
  SUBB A, CAPTURE_0
  MOV PHASE, A
  MOV A, CAPTURE_1+1
  SUBB A, CAPTURE_0+1
  MOV PHASE+1, A
  CLR FLAG_0
  JMP EXIT

MOD1_LEADING:
  MOV A, CAPTURE_0
  SUBB A, CAPTURE_1
  MOV PHASE, A
  MOV A, CAPTURE_0+1
  SUBB A, CAPTURE_1+1
  MOV PHASE+1, A
  CLR FLAG_1

EXIT:
  POP PSW
  POP ACC
  RETI

END
Listing 5. Software Timer

Software Timer mode which interrupts every 20 msec with Fosc = 12 MHz.

; Initialize PCA timer
PCA_INIT:  MOV CMOD, #00H
           MOV CH, #00
           MOV CL, #00
           MOV CCAPM0, #49H
           MOV CCAPOL, #LOW(20000)
           MOV CCAP0H, #HIGH(20000)
           SETB EC
           SETB EA
           SETB CR

; Software Timer mode with interrupt
; Write to low byte first
; Enable PCA interrupt
; Turn PCA timer on

; Test program only
WAIT:    JMP $      ; Wait for PCA interrupt
          JMP WAIT
          ;
          ; This code assumes Module 0 is the only module being used. If
          ; other PCA module's are being used, software must check which
          ; module's event flag caused the interrupt.

PCA_INTERRUPT:

; Clear module 0's event flag
CLR CCF0
PUSH ACC
PUSH PSW
CLR EA
MOV A, #LOW(20000)
ADD A, CCAP0L
MOV CCAP0L, A
MOV A, #HIGH(20000)
ADD A, CCAP0H
MOV CCAP0H, A
SETB EA
POP PSW
POP ACC
RETI

END
Listing 6: High Speed Output (without interrupt)

$nomod51
$sosymbols
$nolist
$include (reg252.pdf)
$list

HSO mode without PCA interrupt. Maximum frequency output = 30.5 Hz
at Fosc = 16 MHz.

ORG 0000H
JMP PCA_INIT

; Initialize PCA timer
PCA_INIT: MOV CMOD, #02H ; Input to PCA timer = 1/4 x Fosc
MOV CH, #00
MOV CL, #00

; HSO Mode without interrupt enabled
MOV CCAPMO, #4CH
MOV CCAPOL, #0FFH
MOV CCAPOH, #0FFH

; P1.3 will toggle every 65,536 counts
; or 16.4 msec at Fosc = 16 MHz
; Period = 30.5 Hz

SETB CR ; Turn PCA timer on

END
Listing 7. High Speed Output (with interrupts)

```
$nomod51
$nosymbols
$nolist
$include (reg252.pdf)
$list

HSO mode with variable frequency. This example outputs a 1KHz signal with Fosc = 16 MHz.

ORG 0000H
JMP PCA_INIT

ORG 0003H
JMP PCA_INTERRUPT

; Initialize PCA timer
PCA_INIT:  MOV CMOD, #02H ; Input to PCA timer = 1/4 x Fosc
        MOV CH, #00
        MOV CL, #00
        MOV CCAPM0, #4DH ; HSO mode with interrupt enabled
        MOV CCAPOL, #LOW(1000) ; t = 1000 arbitrary
        MOV CCAPOH, #HIGH(1000)
        CLR P1.3

; SETB EC ; Enable PCA interrupt
        SETB EA ; Turn PCA timer on

; Test program only
WAIT:    JMP $ ; Wait for PCA interrupt

; This code assumes Module 0 is the only module being used. If other PCA module's are being used, software must check which module's event flag caused the interrupt.
PCA_INTERRUPT:  CLR CCF0 ; Clear module 0's event flag
        PUSH ACC
        PUSH PSW
        CLR EA
        MOV A, #LOW(2000) ; 16-bit add
        ADD A, CCAPOL
        MOV CCAPOL, A
        MOV A, #HIGH(2000)
        ADDC A, CCAPOL
        MOV CCAPOL, A

        SETB EA
        POP PSW
        POP ACC
        RETI

END
```
Listing 8. High Speed Output (Single Pulse)

;nomod51
$nosymbols
$nolist
$include (reg252.pdf)
$first

; HSO mode generates a single pulse width of 20 usecs with Fosc = 16 MHz.

ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

; Initialize PCA timer
PCA_INIT: MOV CMOD, #02H ; Input to PCA timer = 1/4 x Fosc
MOV CH, #00
MOV CL, #00

MOV CCAPM0, #4DH ; HSO mode with interrupt enabled
MOV CCAPOL, #LOW(1000)
MOV CCAPOL, #HIGH(1000)
CLR P1.3

SETB EC ; Enable PCA interrupt
SETB EA ; Turn PCA timer on

; Test program only
WAIT: JMP $ ; Wait for PCA interrupt
JMP WAIT

; This code assumes Module 0 is the only module being used. If other PCA module's are being used, software must check which module's event flag caused the interrupt.

PCA_INTERRUPT:
CLR CCF0 ; Clear module 0's event flag
JNB P1.3, DONE

PUSH ACC
PUSH PSW
CLR EA ; Hold off interrupts
MOV A, #LOW(80)
ADD A, CCAPOL
MOV CCAPOL, A ; 16-bit add
MOV A, #HIGH(80)
ADDCC A, CCAPOL
MOV CCAPOL, A
SETB EA
POP PSW
POP ACC
RETI

DONE: MOV CCAPM0, #00H ; Disable HSO mode
RETI

END
Listing 9. Watchdog Timer

```
$nomod51
$symbols
$notlist
$include (reg252.pdf)
$lst

ORG 0000H
JMP PCA_INIT

; Initialize PCA timer
PCA_INIT:
   MOV CMOD, $00H
   MOV CH, $00
   MOV CL, $00

   MOV CCAPM4, $4CH
   MOV CCAP4L, $0FFH
   MOV CCAP4H, $0FFH

   ORL CMOD, $40H

   SETB CMOD

; Input to PCA timer = 1/12 x Fosc
; Module 4 in compare mode
; Write to low byte first
; Before PCA timer counts up to $FFFF Hex,
; these compare values must be changed
; Set the WDTE bit to enable watchdog timer
; Turn PCA timer on

; Test program only
START:
   MOV R1, $120D
   MOV R0, $0FFH

   DJNZ R0, $
   DJNZ R1, MAIN
   CALL WATCHDOG
   JMP START

; Delay for approx. 60 msec
; Check that watchdog never causes a reset

WATCHDOG:
   CLR EA
   MOV CCAP4L, $00H
   MOV CCAP4H, $00H
   SETB EA
   RET

; Hold off interrupts
; Next compare value is within
; 255 counts of the current PCA
; timer value

END
```

2-281
Listing 10. Pulse Width Modulator

$nomod51
$symbols
$nullist
$include (reg252.pdf)
$list

PWM mode – Maximum frequency output = 15.6 KHz with Fosc = 16 Mhz.

ORG 0000H
JMP PCA_INIT

; Initialize PCA timer
PCA_INIT: MOV CMOD, #02H ; Input to PCA timer = 1/4 x Fosc
MOV CH, #00 ; At 16 MHz, frequency = 15.6 KHz
MOV CL, #00

; MOV CCAPM0, #42H ; PWM Mode
MOV CCAP0L, #00H ; Write to low byte first
MOV CCAP0H, #128D ; 50 percent duty cycle
SETB CR ; Turn PCA timer on

END
APPENDIX B
Duty Cycle Calculation

$DEBUG

SHORT_DIVISION SEGMENT CODE

EXTRN DATA(PULSE_WIDTH, PERIOD, DUTY_CYCLE)
PUBLIC DUTY_CYCLE_CALCULATION

RSEG SHORT_DIVISION

DUTY_CYCLE_CALCULATION

CALCULATES DUTY_CYCLE = PULSE_WIDTH / PERIOD

Inputs to this routine are 16-bit pulse width and period measurements of a rectangular waveform. The output is a 9-bit BCD number representing the duty cycle of the waveform. The low 8 bits of the result are returned in DUTY_CYCLE. The 9th bit is the carry bit in the PSW. If the duty cycle is between 0 and 99 percent, the carry bit is 0 and DUTY_CYCLE contains the two BCD digits representing the duty cycle as a percent. If the duty cycle is 100 percent, the carry bit is 1 and DUTY_CYCLE contains 0.

INPUTS: PULSE_WIDTH 2 bytes in externally defined DATA
(low byte at PULSE_WIDTH, high byte at PULSE_WIDTH+1)

PERIOD 2 bytes in externally defined DATA
(low byte at PERIOD, high byte at PERIOD+1)

OUTPUT: DUTY_CYCLE 1 byte in externally defined DATA

VARIABLES AND REGISTERS MODIFIED:

PULSE_WIDTH, DUTY_CYCLE
ACC, B, PSW, R2, R3

ERROR EXIT: Exit with OV = 1 indicates PULSE_WIDTH > PERIOD.

DUTY_CYCLE_CALCULATION:

MOV A,PERIOD+1
CJNE A,PULSE_WIDTH+1,NOT_EQUAL
MOV A,PERIOD
CJNE A,PULSE_WIDTH,NOT_EQUAL
EQUAL:
  SETB C
  MOV DUTY_CYCLE,#0
  CLR OV
  RET

NOT_EQUAL:
  JNC CONTINUE
  SETB OV
  RET

CONTINUE:
  MOV R2,#8
  MOV DUTY_CYCLE,#0
  MOV R3,#0

TIMES_TWO:
  MOV A,PULSE_WIDTH
  RLC A
  MOV PULSE_WIDTH,A
  MOV A,PULSE_WIDTH+1
  RLC A
  MOV PULSE_WIDTH+1,A
  MOV A,R3
  RLC A
  MOV R3,A

COMPARE:
  CJNE R3,#0,DONE
  CJNE A,PERIOD+1,DONE
  CJNE A,PERIOD,DONE

DONE:
  CPL C

BUILD_DUTY_CYCLE:
  MOV A,DUTY_CYCLE
  RLC A
  MOV DUTY_CYCLE,A
  JNB ACC.0,LOOP_CONTROL

SUBTRACT:
  MOV A,PULSE_WIDTH
  SUBB A,PERIOD
  MOV PULSE_WIDTH,A
  MOV A,PULSE_WIDTH+1
  SUBB A,PERIOD+1
  MOV PULSE_WIDTH+1,A
  MOV A,R3
  SUBB A,#0
  MOV R3,A

LOOP_CONTROL:
  DJNZ R2,TIMES_TWO
FINAL_TIMES_TWO:
  MOV A, PULSE_WIDTH
  RLC A
  MOV PULSE_WIDTH, A
  MOV A, PULSE_WIDTH+1
  RLC A
  MOV PULSE_WIDTH+1, A
  MOV A, R3
  RLC A
  MOV R3, A

FINAL_COMPARE:
  CJNE R3, #0, FINAL_DONE
  MOV A, PULSE_WIDTH+1
  CJNE A, PERIOD+1, FINAL_DONE
  MOV A, PULSE_WIDTH
  CJNE A, PERIOD, FINAL_DONE

FINAL_DONE:
  JC CONVERT_TO_BCD
  MOV A, DUTY_CYCLE
  ADD A, #1
  MOV DUTY_CYCLE, A
  JNC CONVERT_TO_BCD
  CLR OV
  RET

CONVERT_TO_BCD:
  MOV A, DUTY_CYCLE
  MOV B, #10
  MUL AB
  XCH A, B
  SWAP A
  MOV DUTY_CYCLE, A
  MOV A, #10
  MUL AB
  XCH A, B
  ORL DUTY_CYCLE, A
  MOV A, #10
  MUL AB
  MOV A, B
  CJNE A, #5, TEST
  JC OUT_CYC
  MOV A, DUTY_CYCLE
  ADD A, #1
  DA A
  MOV DUTY_CYCLE, A
  OUT: RET

END
A map of the Special Function Register (SFR) space is shown in Table A1. Those registers which are new or have new bits added for the 83C51FA and 83C51FB have been **boldfaced**.

Note that not all of the addresses are occupied. Unoccupied addresses are not implemented on the chip. Read accesses to these addresses will in general return random data, and write accesses will have no effect.

User software should not write 1s to these unimplemented locations, since they may be used in future 8051 family products to invoke new features. In that case the reset or inactive values of the new bits will always be 0, and their active values will be 1.

### Table A1. Special Function Register Memory Map and Values After Reset

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>CH</td>
<td>00000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>F7</td>
</tr>
<tr>
<td></td>
<td>CCAP0H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP1H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP2H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP3H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP4H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0 * B</td>
<td></td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E8</td>
<td>CL</td>
<td>00000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>EF</td>
</tr>
<tr>
<td></td>
<td>CCAP0L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP1L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP2L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP3L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAP4L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0 * ACC</td>
<td></td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E7</td>
</tr>
<tr>
<td>D8</td>
<td>CCON</td>
<td>00X000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>DF</td>
</tr>
<tr>
<td></td>
<td>CMOD</td>
<td>00XX0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAPM0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAPM1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAPM2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAPM3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CCAPM4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0 * PSW</td>
<td></td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D7</td>
</tr>
<tr>
<td>C8</td>
<td>T2CON</td>
<td>00000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>CF</td>
</tr>
<tr>
<td></td>
<td>T2MOD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RCAP2L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RCAP2H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TL2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TH2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C7</td>
</tr>
<tr>
<td>B8 * IP</td>
<td></td>
<td>00X000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>BF</td>
</tr>
<tr>
<td></td>
<td>SADEN</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B0 * P3</td>
<td></td>
<td>11111111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B7</td>
</tr>
<tr>
<td>A8 * IE</td>
<td></td>
<td>00000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>AF</td>
</tr>
<tr>
<td></td>
<td>SADDR</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A0 * P2</td>
<td></td>
<td>11111111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A7</td>
</tr>
<tr>
<td>98</td>
<td>SCON</td>
<td>00000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>9F</td>
</tr>
<tr>
<td></td>
<td>SBUF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>P1</td>
<td>11111111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97</td>
</tr>
<tr>
<td>88 * TCON</td>
<td></td>
<td>00000000</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>8F</td>
</tr>
<tr>
<td></td>
<td>TMOD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TL0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TL1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TH0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TH1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80 * P0</td>
<td></td>
<td>11111111</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>XXXXXXXX</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>SP</td>
<td>00000011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DPL</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DPH</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>PCON</em>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* = Found in the 8051 core (See 8051 Hardware Description in the Embedded Controller Handbook for explanations of these SFRs).

** = See description of PCON SFR. Bit PCON.4 is not affected by reset.

X = Undefined.
Small DC Motor Control

JAFAR MODARES
ECO APPLICATIONS
INTRODUCTION

This application note shows how an 83C51FA can be used to efficiently control DC motors with minimum hardware requirements. It also discusses software implementation and presents helpful techniques as well as sample code needed to realize precision control of a motor.

There is also a brief overview of the new features of the 83C51FA. This new feature is called the Programmable Counter Array (PCA) and is capable of delivering Pulse Width Modulated signals (PWM) through designated I/O pins.

It is assumed that the reader is familiar with the MCS-51 architecture and its assembly language. For more information about the 8051 architecture and the PCA, refer to the Embedded Controller Handbook Volume 1 (order no. 210918-006).

This document will not discuss stepper motors or motor control algorithms.

DC MOTORS

DC motors are widely used in industrial and consumer applications. In many cases, absolute precision in movement is not an issue, but precise speed control is. For example, a DC motor in a cassette player is expected to run at a constant speed. It does not have to run for precise increments which are fractions of a turn and stop exactly at a certain point.

However, some motor applications do require precise positioning. Examples are high resolution plotters, printers, disk drives, robotics, etc. Stepper motors are frequently used in those applications. There are also applications which require precise speed control along with some position accuracy. Video recorders, compact disk drives, high quality cassette recorders are examples of this category.

By controlling DC motors accurately, they can overlap many applications of stepper motors. The cost of the control system depends on the accuracy of the encoder and the speed of the processor.

The 83C51FA can control a DC motor accurately with minimum hardware at a very low cost. The microcontroller, as the brain of a system, can digitally control the angular velocity of the motor, by monitoring the feedback lines and driving the output lines. In addition it can perform other tasks which may be needed in the application.

Almost every application that uses a DC motor requires it to reverse its direction of rotation or vary its speed. Reversing the direction is simply done by changing the polarity of the voltage applied to the motor. Figure 1 shows a simplified symbolic representation of a driver circuit which is capable of reversing the polarity of the input to the motor.

![Figure 1. Reversible Motor Driver Circuit](270622-1)

Varying the speed requires changing the voltage level of the input to the motor, and that means changing the input level to the motor driver. In a digitally-controlled system, the analog signal to the driver must come from some form of D/A converter. But adding a D/A converter to the circuit adds to the chip count, which means more cost, higher power consumption, and reduced reliability of the system.

The other alternative is to vary the pulse width of a digital signal input to the motor. By varying the pulse width the average voltage delivered to the motor changes and so does the speed of the motor. A digital circuit that does this is called a Pulse Width Modulator (PWM). The 83C51FA can be configured to have up to 5 on-board pulse width modulators.

THE 83C51FA

The 83C51FA is an 8-bit microcontroller based on the 8051 architecture. It is an enhanced version of the 87C51 and incorporates many new features including the Programmable Counter Array (PCA).

Included in the Programmable Counter Array is a 16-bit free running timer and 5 separate modules. The PCA timer has two 8-bit registers called CL (low byte) and CH (high byte), and is shared by all modules. It can be programmed to take input from four different sources. The inputs provide flexibility in choosing the count rate of the timer. The maximum count rate is 4 MHz (1/4 of the oscillator frequency).

Some of the port 1 pins are used to interface each module and the timer to the outside world. When the port pins are not used by the PCA modules, they may be used as regular I/O pins.

The modules of the PCA can be programmed to perform in one of the following modes: capture mode,
compare mode, high speed output mode, pulse width modulator (PWM) mode, or watchdog timer mode (only module 4).

Every module has an 8-bit mode register called CCAPMn (Figure 2), and a 16-bit compare/capture register called CCAPnL & CCAPnH, where n can be any value from 0 to 4 inclusive. By setting the appropriate bits in the mode register you can program each module to operate in one of the aforementioned modes.

### Table: CCAPMn

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ECOM</td>
<td>CAPPn</td>
<td>CAPNn</td>
<td>MATn</td>
<td>TOGn</td>
<td>PWMn</td>
<td>ECCFn</td>
</tr>
</tbody>
</table>

**CCAPMn**

**ECOMn** — Enables the comparator function. Must be set for functions which require comparing of the compare/capture registers with the 16-bit timer, i.e., software timer, high-speed output, watchdog timer, and PWM.

**CAPPn** — Capture on positive edge of signal.

**CAPNn** — Capture on negative edge of signal.

**MATn** — Find a match between the capture/compare and 16-bit timer.

**TOGn** — Toggle I/O pin upon a match between capture/compare registers and 16-bit timer.

**PWMn** — Generate PWM on I/O pin upon a match between the low byte of capture/compare and the low byte of PCA timer.

**ECCFn** — Enables compare/capture flag CCFn in the CCON register to generate an interrupt.

*Figure 2. CCAPMn Register*

When a module is programmed in capture mode, an external signal on the corresponding port pin will cause a capture of the current value of the 16-bit timer. By setting bits CAPPn or CAPNn or both, the module can be programmed to capture on the rising edge, falling edge, or either edge of the signal. If enabled, an interrupt is generated at the time of capture.

When module is to perform in one of the compare modes (software timer, high speed output, watchdog timer, PWM), the user loads the capture/compare registers with a calculated value, which is compared to the contents of the 16-bit timer, and causes an event as soon as the values match. It can also generate an interrupt.

PWM is one of the compare modes and is the only one which uses only 8 bits of the capture/compare register. The user writes a value (0 to FFH) into the high byte (CCAPnH) of the selected module. This value is transferred to the lower byte of the same module and is compared to the low byte of the PCA timer. While CL < CCAPnL the output on the corresponding pin is a logic 0. When CL > CCAPnL, the output is a logic 1.

In this application note we will see how a module can be programmed to perform as a PWM to control the speed and direction of a DC motor.

### SETTING UP THE PCA

The 83C51FA has several Special Function Registers (SFRs) that are unknown to ASM51 versions before 2.4. The names of these SFRs must be defined by DATA directive or be defined in a separate file and be included at the time of compilation. Such a file has already been created and is included in the ASM51 package version 2.4.

Two special function registers are dedicated to the PCA timer to allow mode selection and control of the timer. These registers are CCON and CMOD and are shown in figure 3. CCON contains the PCA timer ON/OFF bit (CR), timer rollover flag (CF) and module flags (CCFn). Module flags are used to determine which module causes the PCA interrupt.

*Figure 3. CCON and CMOD Registers*

First the clock source for the PCA timer must be defined. The 16 bit timer may have one of four sources for its input. These sources are: osc freq/4, osc freq/12, timer 0 overflow, and external clock.

Two bits in the CMOD register are dedicated to selecting one of the sources for the PCA timer input. They are bits 1 and 2 of CMOD which are called CPS0 and CPS1. CMOD is not bit addressable, thus the value
must be loaded as a byte. Figure 4 shows all the sources and the corresponding values of CPS0 and CPS1.

<table>
<thead>
<tr>
<th>CPS1</th>
<th>CPS0</th>
<th>TIMER INPUT SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Internal clock, Fosc/12</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Internal clock, Fosc/4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Timer 0 overflow</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>External clock (input on P1.2)</td>
</tr>
</tbody>
</table>

**Figure 4. Timer Input Source**

Next the appropriate module must be programmed as a PWM. As it was noted earlier, the 8-bit mode register for each module is called CCAPMn (see figure 2). Bit 1 of each register is called PWMn. This bit along with ECOMn (bit 6 of the same register) must be set to program the module in the PWM mode. PWM is one of the compare functions of the PCA, and ECOMn enables the compare function. Thus, the hex value that must be loaded into the appropriate CCAPMn register is 42H.

Now that the module is programmed as a PWM, a value must be loaded in the high byte of the compare register to select the duty cycle. The value can be any number from 0 to 255. In the 83C51FA loading 0 in the CCAPnH will yield 100% duty cycle, and 255 (OFFh) will generate a 0.4% duty cycle. See figure 5.

The next step is to start the PCA timer. The bit that turns the timer on and off is called CR and is bit 6 of the CCON register (Figure 3). Since this register is bit addressable, you can use bit instructions to turn the timer on and off.

In the following example module 2 has been selected to provide a PWM signal to a motor driver. An external clock will be provided for the timer input, so the value that needs to be loaded into CMOD is 06H.

**HARDWARE REQUIREMENT**

When using an 83C51FA, very little hardware is required to control a motor. The controller can interface to the motor through a driver as shown in figure 6.

```
MOV CMOD, #06 ; timer input external
MOV CCAPM2, #42H ; put the module in PWM mode.
MOV CCAP2H, #0 ; 0 provides 100% duty cycle (5V)
SETB CR ; turn timer on
...
END
```
This configuration, a closed loop circuit, takes up only three I/O pins. The line controlling direction can be a regular port pin but the speed control line must be one of the port 1 pins which corresponds to a PCA module selected for PWM. Depending on how the feedback is generated and processed, it could be connected to a regular I/O, an external interrupt, or a PCA module. Feedback is discussed in more detail in the feedback section of this application note.

The diagram in Appendix A is an example of a DC motor circuit which has been built and bench-tested.

**DRIVER CIRCUIT**

Although some DC motors operate at 5 volts or less, the 83C51FA can not supply the necessary current to drive a motor directly. The minimum current requirements of any practical motor is higher than any microcontroller can supply. Depending on the size and ratings of the motor, a suitable driver must be selected to take the control signal from the 83C51FA and deliver the necessary voltage and current to the motor.

A motor draws its maximum current when it is fully loaded and starts from a stand still condition. This factor must be taken into account when choosing a driver. However, if the application requires reversing the motor, the current demand will even be higher. As the motor's speed increases, it's power consumption decreases. Once the speed of a motor reaches a steady state, the current depends on the load and the voltage across the motor.

Standard motor drivers are available in many current and voltage ratings. One example is the L293 series which can output up to 1 ampere per channel with a supply voltage of 36 V. It has separate logic supply and takes logical input (0 or 1) to enable or disable each channel. There are four channels per device. The L293D also includes clamping diodes needed for protecting the driver against the back EMF generated during the reversing of motor.

**NOISE CONSIDERATIONS**

Motors generate enough electrical noise to upset the performance of the controller. The source of the noise could be from the switching of the driver circuits or the motor itself. Whatever the cause of the noise may be, it must be isolated or bypassed.

Isolating the microcontroller from the driver circuit is helpful in keeping the noise limited.

Bypass capacitors help a great deal in suppressing the noise. They must be added to the power and ground (Figure 7 diagram a), on the driver circuit (diagram b), on the motor terminals (diagram c), and on the 83C51FA (diagram d). The capacitors must be as close to the component as possible. In fact the best location is under the chip or on top of it if packaging allows. The diagrams in figure 7 show the location and some typical values for the bypass capacitors.
OPEN LOOP & CLOSED LOOP SYSTEMS

There are two types of motor control systems: open loop and closed loop.

In the open loop system the controller outputs a signal to turn the motor on/off or to change the direction of the rotation based on an input that does not come from the motor. For example, the position of a manual or timer switch becomes the input to the controller, which varies the input to the motor. In another case, the controller may take input from data tables in the program to run, vary the speed, reverse direction, or stop the motor.

Closed loop systems can use one or more of the above mentioned examples for the open loop system, plus at least one feedback signal from the motor. The feedback signal provides such information as speed, position, and/or direction of motion.

Many applications require that a motor run at a constant speed. The controller has to continuously make adjustments to keep the speed within the limits. In some cases the speed of the motor is synchronized to another motor or moving part of the system.

Depending on the type of feedback signal, the 83C51FA may have to use other modules of the PCA along with other on-chip peripherals such as Timer/Counters, Serial Port, and the interrupt system to precisely control a DC motor.

The example in the following section uses one PCA module to generate PWM, and another module (in capture mode) to receive feedback from a DC motor.

FEEDBACK

The feedback comes from a sensing device which can detect motion. The sensing device may be an optical encoder, infrared detector, Hall effect sensor, etc. Depending on the application, one or more of the above mentioned sensing devices may be suitable.

The optical sensors should be encapsulated for better reliability. If they are not enclosed, factors such as ambient light, dust, and dirt can lessen their sensitivity.

Hall effect sensors are insensitive to any type of light. They change logic levels going into and coming out of a magnetic field. The sensing device is normally mounted
on some stationary part of the system and the magnet is installed on the rotating part. The potential problem with the Hall effect sensors are that if the gap between the magnet and the sensing device is too big, the sensing device may not be affected by the magnetic field. Also the number of magnets is limited which means fewer feedback pulses will be provided.

Whatever the means of sensing, the result is a signal which is fed to the controller. The 83C51FA can use the feedback signal to determine the speed and position of the motor. Then it can make adjustments to increase or decrease the speed, reverse the direction, or stop the motor.

In the following example module 3 of PCA is set up to perform in the capture mode. In this mode module 3 will receive feedback signals from a Hall effect transistor fixed behind a wheel which is mounted on the shaft of a DC motor. Two magnets are embedded on this wheel in equal distances from each other (180 degrees apart). Every time that the Hall effect transistor passes through the magnetic field, it generates a pulse.

The signal is input to P1.6 which is the external interface for module 3 of the PCA. In this example, module 3 is programmed to capture on the rising edge of the input signal. The time between the two captures corresponds to \( \frac{1}{2} \) of a revolution. Thus, two consecutive captures can provide enough information to calculate the speed of the motor as explained in the next paragraph. By storing the value of the capture registers each time, and comparing it to its previous value, the controller can constantly measure and adjust the speed of the motor. Using this method one can run a motor at a precise speed, or synchronize it to another event.

In the PCA interrupt service routine, each capture value is stored in temporary locations to be used in a subtract operation. Subtracting the first capture from the second one will yield a 16-bit result. The resultant value, which will be referred to as “Result” in the rest of this document, is in PCA timer counts. An actual RPM can be calculated from Result. Although the 83C51FA can do the calculation, it would be much faster to provide a lookup table within the code. The table will contain values which have been calculated for a possible range of Results.

The following code is an example of how to measure the period of a signal input to module 3 of the 83C51FA. The diagram in figure 8 shows how the period corresponds to the rotation of the wheel. In the diagram “T” is the period and “t” is the time that the magnet is passing in front of the Hall effect transistor.

![Diagrams of magnets and Hall effect transistor](image)

Figure 8. The Output Waveform of the Hall Effect Transistor as it goes Through the Magnetic Field
SOFTWARE/CPU OVERHEAD

It takes the 83C51FA no more than 250 bytes of code to control a DC motor. That is to run the motor at various speeds, monitor the feedback, use electrical braking, and even run it in steps. However, the CPU time spent on the above tasks can add up to 70 to 75% of the total time available (clock frequency 12 MHz).

The section of software which turns the motor on and off, or sets the speed is very short. In fact, all of that can be done in less than 30 instructions. Thus, in an open loop system, the controller spends an insignificant amount of time on controlling the motor. However, in a closed loop system the controller has to continuously monitor the speed and adjust it according to the program and the feedback.

The rest of this section talks about electrical braking, stepping a DC motor, and offers examples of code to implement these techniques.
ELECTRICAL BRAKING

Once a DC motor is running, it picks up momentum. Turning off the voltage to the motor does not make it stop immediately because the momentum will keep it turning. After the voltage is shut off, the momentum will gradually wear off due to friction. If the application does not require an abrupt stop, then by removing the driving voltage, the motor can be brought to a gradual stop.

An abrupt stop may be essential to an application where the motor must run a few turns and stop very quickly at a predetermined point. This could be achieved by electrical braking.

Electrical braking is done by reversing the direction of the motor. In order to run in reverse direction, the motor has to stop first, at which time the driving voltage is eliminated so that the motor does not start in the new direction. Therefore the length of time that the reversing voltage is applied must be precisely calculated to ensure a quick stop while not starting it in the reverse direction.

There is no simple formula to calculate when to start, and how long to maintain braking. It varies from motor to motor and application to application. But it can be perfected through trial and error.

In a closed loop system, the feedback can be used to determine where or when to start braking and when to discontinue.

During the electrical braking, or any time that the motor is being reversed, it draws its maximum current. To a motor which is turning at any speed, reversing is a heavy load. The current demand of a motor, when it has been reversed, is much higher than when it has just been powered on.

The following shows a code sample for electrical braking on a DC motor. The code is designed for the hardware shown in Appendix A. The subroutine DELAY provides the period that the reverse voltage is applied to the motor. The code for this subroutine is available in the TIME DELAYS section of this document.

```
BEGIN:
  MOV CMOD,#0 ; SET PCA TIMER INPUT fOSC/12.
  MOV CCAPM1,#42H ; SETTING THE MODULE TO PWM MODE.
  SETB CR ; PCA TIMER RUN.

; DRIVE MOTOR CLOCKWISE
  CLR Pl.0 ; Pl.0 AND THE PWM OF MODULE 1-
  MOV CCAP1H,#00 ; CONTROL THE SPEED AND DIRECTION.

; 00 IN THIS REGISTER PUTS OUT MAX PWM (LOGICAL 1)
  CALL STOP_MOTOR

STOP_MOTOR:
  SETB Pl.0 ; REVERSING THE MOTOR.
  MOV CCAP1H,#0FFH
  CALL DELAY ; WAITING FOR 0.5 SECOND.
  CLR Pl.0 ; REDUCING VOLTAGE TO 0.
  RET ; RETURN FROM SUBROUTINE.
```

2-295
STEPPEING A DC MOTOR

Using the 83C51FA, it is possible to run a simple DC motor in small steps. The resolution of the steps will be as high as the resolution of the encoder. If this resolution is sufficient, here is a technique to run a DC motor in steps.

Using a gear box to gear down the motor will increase the resolution of steps. However, putting too much load through the gears will cause sluggish starts and stops.

Electrical braking is used in order to stop the motor at each step. Therefore, the routine that runs the motor in steps will consist of turning it on with full force, waiting for certain period, and stopping it as fast as possible. The wait period depends on the number of steps per revolution.

As the steps and the intervals between them become smaller, the average current demand of the motor increases. This is because the motor is operated at its maximum torque condition every time it starts to rotate and every time it is reversed for electrical braking.

The following code sample shows a continuous loop which runs the motor in steps. The number of steps per revolution depends on the duration of the delay generated by DELAY subroutine. Subroutine WAIT provides the time between the steps.

Subroutine DELAY is the period of time that the motor is kept in reverse. This period must be determined through trial and error for each type of motor and system.

TIME DELAYS

While the 83C51FA is controlling a motor it must frequently wait for the motor to move to certain position before it can proceed with the next task. For example, in the case of electrical braking when the controller reverses the polarity of voltage across the motor, depending on the type, size, and the speed of the motor, it may have up to a second of CPU time before it will turn the motor off.

The wait may be implemented in different ways. Any of the Timer/Counters or unused PCA modules could be utilized to provide accurate timing. The advantage in using the timers is that while the timer is counting, the processor can be taking care of some other tasks. When the timer times out and generates an interrupt the processor will go back and continue servicing the motor.

If there are no timers or PCA modules available for this purpose, a software timer maybe set up by decrementing some of the internal registers. In this method the processor will be tied up counting up or down and will not be able to do anything else. An example of such a timer is:

```
LOOP:
  CLR P1.0 ; SET DIRECTION CLOCKWISE
  MOV CCAP1H,#0 ; MAX PWM

The above instruction sets the motor running clockwise. The controller can be doing other tasks if need be, or just stay in a wait loop, then stop the motor as shown below.

SETB P1.0 ; REVERSING THE MOTOR.
MOV CCAP1H,#0FFH ;
CALL DELAY ; WAIT FOR IT TO STOP.
CLR P1.0 ; REDUCE VOLTAGE TO 0.
CALL WAIT ; TIME BEFORE NEXT STEP.
JMP LOOP
```

2-296
**DELAY:**

```
MOV R4,#25 ; (decimal)
MOV R5,#255 ; (decimal)
```

**DELAY_LOOP:**

```
DJNZ R5,DELAY_LOOP
DJNZ R4,DELAY_LOOP
RET
```

Subroutine `DELAY` provides approximately 6.4 ms with a 12 MHz clock or 4.8 milliseconds with a 16 MHz clock. The length of this delay can be controlled by loading smaller or larger values to R4 to vary from 260 microseconds up to 65 milliseconds at 12 MHz or 48 milliseconds at 16 MHz oscillator frequency. Larger delays may be obtained by cascading another register and creating an outer loop to this one.

Let us assume that it will take a motor 500 milliseconds to stop from its CW rotation and we are going to use Timer/Counter 0 to provide the wait period. Subroutine `DELAY1` will keep track of this timing. Module 1 of PCA is selected to provide the PWM.

```
ORG OBH
JMP TIMER_INTERRUPT_ROUTINE

CLR P1.0 ; SET DIRECTION CW
MOV CCAP1H,#0 ; MAX PWM

Now the motor is running and the controller can do other tasks. Some typical tasks are called in the following segment.

**BUSY_LOOP:**

```
CALL MONITOR_DISPLAY
CALL SCAN_KEY_BOARD
CALL SCAN_INPUT_LINES
JNB STOP_FLAG,BUSY_LOOP
```

*STOP_FLAG* gets set by a feedback signal and denotes that the motor must stop.

```
SETB P1.0 ; REVERSING THE MOTOR
MOV CCAP1H,#0FFH ;
CALL DELAY ; WAIT TILL MOTOR STOPS
CLR P1.0 ; REDUCE VOLTAGE TO 0
```

```
DELAY1:
SETB EA
SETB ETO ; enable timer 0 interrupt
MOV TLO,#0D8H
MOV TH0,#5E8H
```
To implement a 500 milliseconds delay, timer 0 is used here. In mode 1 timer 0 is a 16-bit timer which takes 65.535 milliseconds at 12 MHz to roll over. Dividing 500 milliseconds to 65.535 shows that timer has to overflow more than 7 times but less than 8 times. How much more than 7 times? The following calculation yields the initial load value of the timer.

\[
500 \div 65.535 = 7.2695 \text{ taking it backward}
\]

\[
65.535 \times 7 = 458.745 \text{ milliseconds}
\]

\[
500 - 458.745 = 41.255 \text{ milliseconds or 41255 microseconds.}
\]

In hexadecimal it is A127H. The initial load value is the complement of this value which is 5ED8H.

CONCLUSION

The 83C51FA with all its on-chip peripherals is a system on one chip. It can simplify the design of a control board and reduce the chip count. Up to 5 DC motors can be controlled while doing other tasks such as monitoring feedback lines, human interfacing (scanning a keyboard, displaying information), and communicating with other processors. The MCS-51 powerful instruction set provides maximum flexibility with minimum hardware.

With its onboard program memory capability, the need for the external EPROM and address latch is eliminated. The 83C51FA can have up to 8K bytes of code and the 83C51FB can have up to 16K bytes of code onboard.

This microcontroller can be used in industrial, commercial, and automotive applications.
Figure A-1 shows a symbolic view of the L293B driver. This driver has 4 channels but only two are shown here. Note the inputs A and B and how they are related to each other. You can input the PWM to either one of the inputs and by toggling the other input start or stop the motor. While running, the PWM input controls the speed. Pin P1.4 corresponds to module 1 of the PCA, and pin P1.0 is used as a regular I/O pin.

Figure A-2 shows the schematic of the motor driver, motor, feedback path, and the supporting components.

![Figure A-1. The L293B Motor Driver](image-url)
All diodes are the same and could be any of the 1N4000 series.

Figure A-2. Full Schematic of a Motor-Control System
Application Techniques for the 83C152 Global Serial Channel in CSMA/CD Mode

BOB JOHNSON
Embedded Control Applications Engineering
INTRODUCTION

The 83C152 is an 80C51BH based microcontroller with DMA capabilities and a high speed, multi-protocol, synchronous serial communication interface called the Global Serial Channel (GSC). The GSC uses packetized data frames that consist of a beginning of frame (BOF) flag, address byte(s), data byte(s), a Cyclic Redundancy Check (CRC), and an End Of Frame (EOF) flag. An example of this type of packet is shown in Figure 1. Most 80C152 users will be familiar with UARTs, another type of serial interface. Figures 1 and 2 compare the two types of frames. The UART uses start and stop bits with a data byte between as shown in Figure 2. The 83C152 retains the standard MCS®-51 UART.

The 83C152 will be referred to as the “C152” throughout this application note to refer to the device. This application note deals with initializing and running the GSC in CSMA/CD mode only. Carrier Sense Multiple Access with Collision Detection (CSMA/CD) is a communication protocol that allows two or more stations to share a common transmission medium by sensing when the link is idle or busy (Carrier Sense). While in the process of transmission, each station monitors its own transmission to identify if and when a collision occurs. When a collision occurs, each station involved in the transmission executes a backoff algorithm and reattempts transmission (Collision Detection). This access method allows all stations an equal chance to transmit its own packet and thus is referred to as a “peer-to-peer” type protocol (Multiple Access). Even in CSMA/CD mode, the user has several variations that can be implemented. Table I summarizes the various CSMA/CD options available. Most of these variations will be discussed in this application note.

![Figure 1. Packetized Frame](image1)

![Figure 2. UART Byte](image2)
Table 1. CSMA/CD Variations Supported by C152

<table>
<thead>
<tr>
<th>CSMA/CD Parameter</th>
<th>Options Supported by Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
<td>8-Bits</td>
</tr>
<tr>
<td></td>
<td>32-Bits</td>
</tr>
<tr>
<td></td>
<td>64-Bits</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td>Hardware</td>
</tr>
<tr>
<td></td>
<td>Software</td>
</tr>
<tr>
<td>Backoff Algorithm</td>
<td>Normal</td>
</tr>
<tr>
<td></td>
<td>Alternate</td>
</tr>
<tr>
<td></td>
<td>Deterministic</td>
</tr>
<tr>
<td>CRC</td>
<td>16-Bit</td>
</tr>
<tr>
<td></td>
<td>32-Bit</td>
</tr>
<tr>
<td>Address Recognition</td>
<td>8-Bit</td>
</tr>
<tr>
<td></td>
<td>16-Bit</td>
</tr>
<tr>
<td></td>
<td>S/W Extendable</td>
</tr>
<tr>
<td>Address Masking</td>
<td>8-Bit</td>
</tr>
<tr>
<td></td>
<td>16-Bit</td>
</tr>
<tr>
<td>Jam Type</td>
<td>D.C.</td>
</tr>
<tr>
<td></td>
<td>CRC</td>
</tr>
<tr>
<td>GSC Servicing</td>
<td>CPU</td>
</tr>
<tr>
<td></td>
<td>DMA</td>
</tr>
<tr>
<td>Data Source (Transmitter)</td>
<td>External RAM</td>
</tr>
<tr>
<td>Data Destination (Receiver)</td>
<td>External RAM</td>
</tr>
<tr>
<td>GSC Interface</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>Buffers</td>
</tr>
<tr>
<td>Baud Rate</td>
<td>1.709 KPBS (minimum)</td>
</tr>
<tr>
<td></td>
<td>2.062 MPBS (maximum)</td>
</tr>
<tr>
<td># Collisions Permitted</td>
<td>0 to 8</td>
</tr>
<tr>
<td># of Slots (Deterministic Only)</td>
<td>1 to 63</td>
</tr>
<tr>
<td>Time Slot</td>
<td>1 to 256 B/Ts</td>
</tr>
<tr>
<td>IFS</td>
<td>2 to 256 B/Ts</td>
</tr>
</tbody>
</table>

In this application note initializing the GSC is covered first. Starting, maintaining, and ending transmissions and receptions will then be discussed. Included in these sections will be how interrupts are generated, the software needed to respond to interrupts, and restarting the process. There are four interrupts used in conjunction with the GSC. They are: Transmit Valid, Transmit Error, Receive Valid, and Receive Error. A complete software example is shown in Appendix A. Included in the software are comments describing what and why certain sections of code are needed.

Figures 3 and 4 are flow charts that show the entire process of using the C152 GSC under CPU or DMA control. Both flow charts begin with initialization which is described in the next section. Each step in the flow charts will be described. In general, the text combines CPU and DMA control of the GSC and discusses pros and cons of each.

These flow charts were created from lab experiments performed with the C152. The purpose of the lab experiments was to implement a CSMA/CD link, over which data could be passed from one station to another. As a source for data to transmit and a method to display the data received, two terminals were used. Connecting two terminals together would not normally be encountered in an actual application. However, connecting two terminals together provided a convenient configuration on which to develop the necessary software. Connecting two terminals also created a base from which the user could implement many different designs utilizing the software provided in Appendix A.

The final experiment consisted of two parts: 1) data received by the UART to be transmitted by the GSC and 2) data received by the GSC to be transmitted by the UART. In both cases a terminal was connected to the UART on each C152 and the GSC was under DMA control. There were eight external 120-byte buffers available. Four buffers were used to store the data received by the UART and four buffers used to store the data received by the GSC.

As data is received from the UART each byte is examined, placed in an external buffer and a counter incremented. Each byte is examined to see if it equals an ASCII “carriage return” (ODH). If a match occurs, the program assumes it is the end of a line and the end of the current buffer. Once a carriage return is detected, a line feed is added and the byte count incremented. The counter is then used to load the byte count register for the appropriate DMA channel. Once a buffer is closed it’s flagged as having data available for the GSC to transmit. If the next buffer was not filled with data waiting to be transmitted by the GSC, it is made available for receiving the next line. Once the GSC transmits the entire packet the buffer is flagged as empty and available for storing new data from the UART.

When a packet is received by the GSC, the data is placed in an external buffer. When the packet ends, the
The number of bytes received is calculated. The current buffer is marked to indicate that the data is ready for output by the UART. The calculated byte count is used to identify how many bytes the UART should send to the terminal. When the UART sends the proper number of bytes, the buffer is made available so that the GSC may store data in it.

This has all been subjected to limited testing in the lab and verified to work with two terminals. The software has only been developed to the point that the terminals may display each other's outgoing messages and no farther. This means that some error conditions are not resolved with the current version of the software. For instance, if two terminals transmit data at approximately the same time, both messages may be displayed, even if the received data occurs within the middle of a sentence being typed. For reasons such as this, the software and hardware presented should not be used for a production product without thorough testing in the actual application.

Figure 3. GSC CPU Flow Chart
Figure 3. GSC CPU Flow Chart (Continued)
Figure 3. GSC CPU Flow Chart (Continued)
Figure 4. GSC DMA Flow Chart
Figure 4. GSC DMA Flow Chart (Continued)
Figure 4. GSC DMA Flow Chart (Continued)
GSC INITIALIZATION

During initialization, user software sets up the hardware in the GSC so that communication may begin and institute the parameters specified by the protocol. This can further be sub-divided into two more sections. The first deals with those items which will vary according to the protocol being implemented, referred to as protocol dependent. The second section deals with those items that need to be accomplished in the same manner regardless of the protocol and are referred to as protocol independent. Table 2 shows those items of initialization which are protocol dependent. Once set up, the items in Table 2 do not have to be repeated when starting a new reception or transmission.

<table>
<thead>
<tr>
<th>Table 2. Protocol Dependent Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>baud rate</td>
</tr>
<tr>
<td>preamble length</td>
</tr>
<tr>
<td>backoff mode (random or deterministic)</td>
</tr>
<tr>
<td>CRC</td>
</tr>
<tr>
<td>interframe space (IFS)</td>
</tr>
<tr>
<td>type of jamming signal used</td>
</tr>
<tr>
<td>slot time</td>
</tr>
<tr>
<td>addressing</td>
</tr>
<tr>
<td>enabling Hardware Based Acknowledge (HBA)</td>
</tr>
</tbody>
</table>

Table 2 introduces two new terms that previous CSMA/CD users may not be familiar with; Hardware Based Acknowledge (HBA) and Deterministic Collision Resolution (DCR). HBA is a method in which the GSC receiver hardware will acknowledge the reception of a valid frame and DCR is a collision resolution algorithm in which the user assigns a specific slot number to each station on the link. HBA will be covered in its own section, located later in this document. For a description on DCR or more information on HBA, please refer to the 83C152 Hardware Description in the 8-bit Embedded Controller Handbook (order # 270645).

Table 3 shows items which are protocol independent. All of the items in Table 3, except for determining how the GSC is controlled, will need to be repeated after each GSC operation, before a reception or transmission starts again.

<table>
<thead>
<tr>
<th>Table 3. Protocol Independent Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearing the collision counter register</td>
</tr>
<tr>
<td>control of the GSC</td>
</tr>
<tr>
<td>initializing DMA (only if used)</td>
</tr>
<tr>
<td>initializing counters and pointers</td>
</tr>
<tr>
<td>enabling the receiver and receive interrupts</td>
</tr>
<tr>
<td>enabling the transmitter and transmit interrupts</td>
</tr>
</tbody>
</table>

INITIALIZATION (PROTOCOL DEPENDENT)

This section deals with those items which are part of initialization which vary according to the protocol being implemented. These parameters will typically be dictated by rules of the protocol or hardware environment. In addition, some parameters will vary according to the software implemented by the programmer. For instance, interframe space (IFS) is one of the parameters dependent on other software developed to implement a protocol with the C152.

BAUD RATE—When initializing the GSC baud rate there are two major considerations. The first is that the GSC baud rate can only be programmed in multiples of 1/8 the oscillator frequency when using the internal baud rate generator as shown in the formula given below. If a 1 MBPS rate is desired, the oscillator frequency must be 16 MHz or 8 MHz. This becomes less critical when the GSC baud rate is much lower than the desired oscillator frequency.

\[
\text{GSC baud rate} = \frac{F_{osc}}{(\text{BAUD} + 1) \times 8}
\]

\[
\text{UART baud rate (Mode 3)} = \frac{(2^{\text{smod}})(F_{osc})}{(256 - \text{TH1}) \times 384}
\]

The second major consideration only matters if the UART is used. In this case, when deciding on GSC baud rate and oscillator frequency the effect on the UART baud rate must be understood. As shown in the formula above, when using a timer in mode 3, baud rates generated for the UART are in multiples of 1/384 the oscillator frequency. This means that standard UART baud rates such as 9600, 2400, 1200, etc. and common GSC baud rates such as 2 MBPS, 1 MBPS, and 640 KBPS, cannot be reached with any single oscillator frequency. This can be worked around with methods such as externally clocking the timers. Externally clocking the GSC cannot be done when CSMA/CD is selected. For instance, the maximum oscillator frequency that can be used to achieve a standard UART baud rate of 9600 is 14.7456 MHz, which works out to a maximum GSC baud rate of 1.8432 MBPS which can be further divided down by multiples of 8. The program example in Appendix A uses these values.
To select a desired baud rate, the Special Function Register BAUD is loaded with an appropriate number according to the previously given formula. For instance:

MOV BAUD,#0 ;selects a baud rate of 1/8 the oscillator frequency
or:
MOV BAUD,#1 ;selects a baud rate of 1/16 the oscillator frequency

at the other extreme:

MOV BAUD,#0FFH ;selects a baud rate of 1/2048 the oscillator frequency

; oscillator frequency 

(7.2K @ 14.7456 MHz)

PREAMBLE LENGTH—A preamble serves four functions in CSMA/CD mode: to provide synchronization for the following frame, to contain the Beginning Of Frame flag (BOF), to let other stations on the link know that the link is being used, and to provide a window where collisions may occur and automatically reattempt transmission (backoff). Figure 5 shows what an eight-bit preamble would look like.

The C152 receiver will synchronize to the first transition and resynchronize on every following transition. For this reason a minimum preamble length can be used. On the C152 the minimum preamble length is 8-bits. However, due to network topography, other devices used, or the protocol being implemented, a larger number of transitions may be required. In these cases the C152 can be programmed for either a 32- or 64-bit preamble.

To select an 8-bit preamble:

GMOD = XXXXX01X

To select a 32-bit preamble:

GMOD = XXXXX10X

To select a 64-bit preamble:

GMOD = XXXXX11X

BACKOFF MODE—The C152 has three types of backoff modes: Normal Backoff, Alternate Backoff, and Deterministic Backoff. Normal backoff and alternate backoff are very similar and the only difference between them is when the slot timer begins counting time slots.

In normal backoff each station randomly chooses a slot based on the number of collisions that have previously occurred. After the idle (EOF) is detected, the interframe space timer and slot time timer begin at the same time. Since all devices are prevented from beginning a transmission during the interframe space, that amount of time is taken away from a device which has chosen slot 0. When a slot time is significantly larger than the interframe space, this should pose no problem as slot 0 will still provide a window for the device to begin transmission. There is a problem when the interframe space is larger than the slot time. In this case, if a device chooses slot 0, it will not be allowed to transmit because the interframe space has not yet expired. This decreases efficiency of the backoff algorithm and reduces bandwidth. Normal backoff should be used when the slot time is greater than the interframe space period.

In alternate backoff, after the idle is detected, only the interframe space timer begins. When the interframe space timer expires, the slot time timer begins. This results in extending the total amount of time spent in the backoff algorithm but preserves the entire amount of time for each slot that may be selected. Alternate backoff is recommended when the slot time is less than or equal to the interframe space period.

The deterministic backoff mode is a new resolution mode introduced by the C152. Deterministic backoff utilizes peer-to-peer communication while in normal transmission mode, and a prioritized or a deterministic algorithm while performing the resolution. Deterministic backoff operates by following standard CSMA rules when attempting to transmit a packet for the first time. However, if a collision is detected each station is
restricted to only transmit during its assigned slot. The slot number is assigned by the user and up to 63 slots are available. A more detailed description on deterministic backoff is in the 80C152 Hardware Description chapter in the 8-bit Embedded Controller Handbook. Deterministic backoff is recommended if there are 64 stations or less in a network and the user wishes to remove the uncertainty that arises when using one of the other two random resolution methods already described. Another reason for using deterministic resolution is if a user wishes to assign a priority to one station’s messages over that of another station’s during the collision resolution period. The user should be aware that most CSMA/CD protocols that already have standards associated with them preclude the use of deterministic backoff.

To select normal backoff:
\[ \text{GMOD} = \text{X0XXXXXX} \]
\[ \text{MYSLOT} = \text{X0XXXXXX} \]

To select alternate backoff:
\[ \text{GMOD} = \text{X1XXXXXX} \]
\[ \text{MYSLOT} = \text{X0XXXXXX} \]

To select deterministic backoff:
\[ \text{GMOD} = \text{X1XXXXXX} \]
\[ \text{MYSLOT} = \text{X1XXXXXX} \]

**CRC—**The C152 offers a choice of two types of CRC. One type of CRC is CRC-CCITT (16-bit) used in HDLC (Reference 1). The second CRC available is named AUTODIN-II (32-bit) which is used in 802.3 (Reference 2). The following formulas give the CRC generating polynomial of each.

- **CRC-CCITT**
  \[ X^{16} + X^{12} + X^5 + 1 \]

- **AUTODIN-II**
  \[ X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^{8} + X^7 + X^5 + X^4 + X^2 + X + 1 \]

The selection of which CRC to use is normally dictated by the protocol being implemented. When selecting a CRC, the user should remember that the CRC length also determines the jam time, which in turn will affect the slot time.

To select the 16-bit CRC:
\[ \text{GMOD} = \text{XXXX0000} \]

To select the 32-bit CRC:
\[ \text{GMOD} = \text{XXXX11XX} \]

**INTERFRAME SPACE—**The interframe space provides a period of time for the receiver and physical medium to fully recover from a previous reception and be prepared to accept a new message. To fulfill these requirements the value programmed into IFS should be greater than or equal to the "turn around" time plus round trip propagation time. "Turn around" time is the amount of time it takes for a receiver to be re-enabled after having just received a previous packet. Calculating worst case turn around time is very complicated when the GSC is under CPU control. This is because the Receive Done bit (RDN), which signifies the end of a received packet, does not generate an interrupt. The user is required to periodically poll Receive Done to ascertain when incoming packets are complete. Since the polling sequence is sometimes altered by interrupts, these delays must also be taken into account when deciding what interframe space will be used. As an alternative, the user could choose to set-up a timer that will periodically poll the receive done bit and give a more reliable idea of what the turn around time will be. This will require that the timer interrupt be assigned a higher priority than any of the other interrupts. Since the RDN bit will be set approximately two bit times after the last CRC bit is received, in some situations it is possible to add a delay to a receive valid interrupt and check Receive Done just prior to leaving the routine. As a last resort a user could ignore the maximum response time and instead pick a number that works most of the time. The only negative result of doing this is that some frames may be missed. If acknowledgements are used, that frame would be retransmitted. However, if acknowledgements are not used, the data would be lost forever.

The programming quantum for interframe space is in bit times where a bit time is equal to 1/baud rate. The only hardware restrictions the C152 places on interframe space is that the number programmed must be even and the maximum value is 256 bit times. Other than that, the user can decide what interframe space value will be used. The interframe space should be the same for all stations on any given network.

To program the interframe space:
\[ \text{IFS} = \text{nmmmmnnn} \]

where nmmmmnnn = number of bit times programmed by the user.

The following two examples show the actual code the C152 will execute in response to a receive interrupt. Only those portions of the code associated with servicing the interrupt are shown. Added to this software, on the left edge, is the number of machine cycles it takes to execute each instruction. With this extra information the required interframe space can be calculated by totaling the number of machine cycles.
The first example gives the flow used for a valid GSC reception and the other example shows the steps taken to service an invalid reception. These examples were created by first implementing a working prototype. Once completed, the software used to service the appropriate interrupt was pulled out, selecting the worst case (longest) flow. Finally, each step was sequentially pieced together to demonstrate how the application services an interrupt. These software fragments are taken from the program in Appendix A.

The total number of machine cycles it takes to service a valid reception (59 cycles) or an invalid reception (115 cycles) is also given. As shown, an invalid reception takes the longest amount of time to service. To 115 cycles we add maximum interrupt latency, which is 9 machine cycles. The total comes out to be 124 machine cycles. It should be mentioned that the typical interrupt latency in the C152 would be about 5 machine cycles.

A 9 machine cycle latency can only occur if the interrupt happens during an access to an interrupt register followed by a multiply or divide instruction and assumes that the receive error interrupt is the only high priority interrupt.

A bit time works out to be 8 oscillator periods (BAUD = 0) in this example. To calculate the number to load into IFS the following formula is used. “12” comes about from the 12 oscillator periods that make up a machine cycle.

\[
IFS = \frac{12 \times (\# \text{ of machine cycles to service the interrupt})}{(\# \text{ of oscillator periods per bit time})}
\]

This works out to be:

\[(12 \times 124)/8 = 186\]

This number should have a guardband added in case minor changes must be made in the routines. Since the only other enabled interrupt is the UART, a small guardband of 10 was used. The interframe space chosen is 196.
Example 1. GSC Receive Valid Service Routine

<table>
<thead>
<tr>
<th>(No. of machine cycles)</th>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>358</td>
<td>002B</td>
<td>359</td>
<td>GSC_RECVALID:</td>
<td></td>
</tr>
<tr>
<td>002B</td>
<td>020568</td>
<td>360</td>
<td>JMP GSC_VALID_REC</td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>GSC_VALID_REC:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>056A</td>
<td>C082</td>
<td>1682</td>
<td>PUSH DPL</td>
<td></td>
</tr>
<tr>
<td>06A</td>
<td>C083</td>
<td>1683</td>
<td>PUSH DPH</td>
<td></td>
</tr>
<tr>
<td>06C</td>
<td>C0E0</td>
<td>1684</td>
<td>PUSH ACC</td>
<td></td>
</tr>
<tr>
<td>056E</td>
<td>C0D0</td>
<td>1685</td>
<td>PUSH PSW</td>
<td></td>
</tr>
<tr>
<td>0570</td>
<td>71B0</td>
<td>1688</td>
<td>CALL NEW_BUFFER2_IN</td>
<td></td>
</tr>
<tr>
<td>1689</td>
<td>NEW_BUFFER2_IN:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03B0</td>
<td>207343</td>
<td>1064</td>
<td>JB GSC_IN_MSB,GSC_IN_2</td>
<td></td>
</tr>
<tr>
<td>1067</td>
<td>GSC_IN_2D_2A:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03F6</td>
<td>20721E</td>
<td>1170</td>
<td>JB GSC_IN_LSB,GSC_IN_2</td>
<td></td>
</tr>
<tr>
<td>1172</td>
<td>GSC_IN_2D:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03F9</td>
<td>2074F6</td>
<td>1175</td>
<td>JB BUF2D_ACTIVE,BUFFER</td>
<td></td>
</tr>
<tr>
<td>03FC</td>
<td>758200</td>
<td>1179</td>
<td>MOV DPL,#LOW (BUF2C_ST</td>
<td></td>
</tr>
<tr>
<td>03FF</td>
<td>758303</td>
<td>1180</td>
<td>MOV DPH,#HIGH (BUF2C_S</td>
<td></td>
</tr>
<tr>
<td>0402</td>
<td>C3</td>
<td>1184</td>
<td>CLR C</td>
<td></td>
</tr>
<tr>
<td>0403</td>
<td>7476</td>
<td>1186</td>
<td>MOV A,#(MAX_LENGTH)</td>
<td></td>
</tr>
<tr>
<td>0405</td>
<td>95F2</td>
<td>1192</td>
<td>SUBB A,BCRL1</td>
<td></td>
</tr>
<tr>
<td>0407</td>
<td>F0</td>
<td>1194</td>
<td>MOVX @DPTR,A</td>
<td></td>
</tr>
<tr>
<td>0408</td>
<td>D275</td>
<td>1197</td>
<td>SETB BUF2C_ACTIVE</td>
<td></td>
</tr>
<tr>
<td>040A</td>
<td>D272</td>
<td>1202</td>
<td>SETB GSC_IN_LSB</td>
<td></td>
</tr>
<tr>
<td>040C</td>
<td>D273</td>
<td>1203</td>
<td>SETB GSC_IN_MSB</td>
<td></td>
</tr>
<tr>
<td>040E</td>
<td>757981</td>
<td>1206</td>
<td>MOV GSC_INPUT_LOW,#LOW</td>
<td></td>
</tr>
<tr>
<td>0411</td>
<td>757803</td>
<td>1207</td>
<td>MOV GSC_INPUT_HIGH,#HI</td>
<td></td>
</tr>
<tr>
<td>0414</td>
<td>020432</td>
<td>1211</td>
<td>JMP NEW_BUF2_IN_END</td>
<td></td>
</tr>
<tr>
<td>1212</td>
<td>NEW_BUF2_IN_END:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0432</td>
<td>8579D2</td>
<td>1253</td>
<td>MOV DARL1,GSC_INPUT_LO</td>
<td></td>
</tr>
<tr>
<td>0435</td>
<td>8578D3</td>
<td>1254</td>
<td>MOV DARH1,GSC_INPUT_HI</td>
<td></td>
</tr>
<tr>
<td>0438</td>
<td>75F300</td>
<td>1258</td>
<td>MOV BCRL1,#0</td>
<td></td>
</tr>
<tr>
<td>043B</td>
<td>75F278</td>
<td>1259</td>
<td>MOV BCRL1,#MAX_LENGTH</td>
<td></td>
</tr>
<tr>
<td>043E</td>
<td>22</td>
<td>1261</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>1263</td>
<td>RETI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'0572</td>
<td>439301</td>
<td>1693</td>
<td>ORL DCON1,#01</td>
<td></td>
</tr>
<tr>
<td>0575</td>
<td>D2E9</td>
<td>1695</td>
<td>SETB GREN</td>
<td></td>
</tr>
<tr>
<td>0577</td>
<td>D0D0</td>
<td>1697</td>
<td>POP PSW</td>
<td></td>
</tr>
<tr>
<td>0579</td>
<td>D0E0</td>
<td>1698</td>
<td>POP ACC</td>
<td></td>
</tr>
<tr>
<td>057B</td>
<td>D083</td>
<td>1699</td>
<td>POP DPH</td>
<td></td>
</tr>
<tr>
<td>057D</td>
<td>D082</td>
<td>1700</td>
<td>POP DPL</td>
<td></td>
</tr>
<tr>
<td>057F</td>
<td>32</td>
<td>1702</td>
<td>RETI</td>
<td></td>
</tr>
</tbody>
</table>

59 TOTAL CYCLES

Example 1. GSC Receive Valid Service Routine
<table>
<thead>
<tr>
<th># of machine cycles</th>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) 0033 0033 020580</td>
<td>362</td>
<td>ORG 33H</td>
<td>364</td>
<td>JMP GSC_ERROR_REC</td>
</tr>
<tr>
<td>(2) 0580 0582 0584 0586</td>
<td>1703</td>
<td>GSC_ERROR_REC:</td>
<td>1735</td>
<td>RCABT_CHECK:</td>
</tr>
<tr>
<td>(2) 0592 059C 059F 05A1</td>
<td>1745</td>
<td>JNB RCABT,OVR_CHECK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2*6) 0278 0279 027B 027C 027D 027F</td>
<td>566</td>
<td>INC_COUNT_LOOP:</td>
<td>588</td>
<td>COUNTER_OVERFLOW:</td>
</tr>
<tr>
<td>(2) 0281 0282</td>
<td>582</td>
<td>RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) 003B 03F6</td>
<td>1063</td>
<td>NEW_BUFFER2_IN:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) 03B0 03F6</td>
<td>1168</td>
<td>GSC_IN_2D_2A:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 2. GSC Receive Error Service Routine
### Example 2. GSC Receive Error Service Routine (Continued)

**JAMMING SIGNAL**—The purpose of a jam is to ensure all stations on a link detect that a collision has occurred and reject that frame. To meet this need, the C152 offers two types of jamming signals. One type of jam is the D.C. jam (Figure 6) and another type is called the CRC (Figure 7) jam. A jam is forced by the TxD pin after a collision is detected but after the preamble ends if the preamble is not yet complete. The D.C. jam forces a constant logic “0” for a period of time equal to the CRC length. The CRC jam takes the CRC calculated up to the point when a collision occurs, complements the CRC, and transmits that pattern. The CRC jam should be used when A.C. coupling is used in a network. A.C. coupling normally implies that pulse transformers or capacitors are used to connect to the serial link. In these types of circuit interfaces, the D.C. jam may not be passed through reliably. One drawback of the CRC jam is that it does not always guarantee that all stations on a link will detect the jamming signal as there are no Manchester code violations inherent in the waveform. The D.C. jam is recommended whenever it can be used since this type of jam will always be detected by forcing Manchester code violations. Some protocols specify a specific type of jam signal that should be used and the user will have to decide if the C152 can fulfill those requirements.

![Figure 6. D.C. Jam](image-url)
To select D.C. jam:
MY SLOT = 1XXXXXXX

To select CRC jam:
MY SLOT = 0XXXXXXX

SLOT TIME—In CSMA/CD networks a slot time should be equal to or larger than the sum of round trip propagation time plus maximum jam time. The slot time is used in the backoff algorithm as a rescheduling quantum. The slot time is programmed in bit times and in the C152 can vary from 1 to 256.

To program the slot time:
SLOTTM = nnnnnnn

ADDRESSING—When discussing the subject of addressing with respect to the C152, the subject should be broken down into three major topics. These topics are: address length, assignment of addresses, and address masking.

Address Length—The C152 gives a user a choice of either 8 or 16 bits of address recognition. To select 8-bit addressing the user must set the AL bit in GMOD to 0. Setting AL to 1 selects 16-bit addressing. Address recognition can be extended with software by examining subsequent bytes for a match. The only part of the GSC hardware that utilizes address length is the receiver. The receiver uses address length to determine when an incoming packet matches a user assigned address. Since transmission of addresses is done under software control, the transmitter does not use the address length bit. All bits following BOF are loaded into RFIFO, including address. The transmit circuitry is involved with addressing only if HBA is used. In this case, when HBA is selected, the transmitter must know whether or not the sending address was even or odd. Even addresses require an acknowledgement back and odd addresses do not.

When transmitting, the user must insert a destination address in the frame to be transmitted. This is done by loading the appropriate address as the first byte or two bytes of data. If a source (sending) address is also to be sent, the user must place that address into the proper position within a packet according to the protocol being implemented.

To select 8-bit addresses:
GMOD = XXX0XXXX

To select 16-bit addresses:
GMOD = XXX1XXXX

Address Assignment—When assigning an address to a station, there are several factors to consider. To begin with, there are four 8-bit address registers in the C152: ADR0, ADR1, ADR2, and ADR3. These registers are initialized to 00 after a valid reset. For this reason it is recommended that no assigned addresses should equal 0. Also, since there are four address registers, a user has a minimum of two addresses which can be assigned to each station when using 16-bit addressing or four addresses when using 8-bit addressing. Those registers not used do not need to be initialized. When using 16-bit addresses ADR1:ADR0 form one 16-bit address and ADR3:ADR2 form a second address. The C152 will always recognize an address consisting of all 1s, which is considered a “broadcast” address. An address consisting of all 1s should not be assigned to any individual station.

There are many methods used to assign addresses. Some suggestions are: reading of a switch, addresses contained in actual program code, assignment by another node, or negotiated with the system. As mentioned earlier, if HBA is being used then the LSB of the address must be 0 when acknowledgements are expect-
Since more than one address can be assigned per station it is possible to use or not use HBA within the same station. This would work by assigning one address that would be even for when acknowledgements are required and another assigned address would be odd for those occasions when acknowledgements are not needed.

To assign an 8-bit address:
\[
ADR_0 = \text{n}nnnnnnn
\]
and optionally:
\[
ADR_1 = \text{x}xxxxxxx
ADR_2 = \text{y}yyyyyyy
ADR_3 = \text{z}zzzzzzzz
\]

To assign a 16-bit address:
\[
ADR_0 = \text{n}nnnnnnn \text{ (lower byte)}
ADR_1 = \text{x}xxxxxxx \text{ (upper byte)}
\]
and optionally:
\[
ADR_2 = \text{y}yyyyyyy \text{ (lower byte)}
ADR_3 = \text{z}zzzzzzzz \text{ (upper byte)}
\]

where xxxxxxxx, yyyyyyyy, zzzzzzzz are addresses to be assigned.

In this example there are 5 nodes (A, B, C, D, and E) with up to 4 common peripherals. The peripherals are: terminals, keyboards, printers, and modems. Assuming 8-bit addressing, a specific address bit is assigned to each peripheral: bit 1 to terminals, bit 2 to keyboards, bit 3 to printers, and bit 4 to modems. Figure 8 shows how this addressing is mapped.

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>BIT 7</th>
<th>BIT 6</th>
<th>BIT 5</th>
<th>BIT 4</th>
<th>BIT 3</th>
<th>BIT 2</th>
<th>BIT 1</th>
<th>BIT 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N.U.</td>
<td>N.U.</td>
<td>N.U.</td>
<td>MODEM</td>
<td>PRINTER</td>
<td>KEYBOARD</td>
<td>TERMINAL</td>
<td>GROUP ADDR</td>
</tr>
<tr>
<td>N.U. = NOT USED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8. Group Addressing Map**

Bit 0 is used to differentiate between group addresses and individual addresses. If bit 0 = 1, then the address is a group address, if bit 0 = 0, then the address is an individual address. This also complies with the HBA requirements if HBA is enabled. Table 4 defines which stations have which peripherals.

| Station A: | Terminal, Keyboard |
| Station B: | Printer, Modem |
| Station C: | Terminal |
| Station D: | Printer |
| Station E: | Terminal, Keyboard, Printer, Modem |

**Table 4. Peripheral Assignment for Example 3**

The next step is to assign each station's address and address mask. These are determined by the attached peripherals. A 1 is placed in the address register bit and address mask register bit if that station has an appropriate device. A 1 in the address register is not used since it is masked out, but will make it easier for a person not familiar with this specific software to follow the program.

<table>
<thead>
<tr>
<th>Address</th>
<th>Address Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT 7</td>
<td>6</td>
</tr>
<tr>
<td>A:</td>
<td>0</td>
</tr>
<tr>
<td>B:</td>
<td>0</td>
</tr>
<tr>
<td>C:</td>
<td>0</td>
</tr>
<tr>
<td>D:</td>
<td>0</td>
</tr>
<tr>
<td>E:</td>
<td>0</td>
</tr>
</tbody>
</table>

**EXAMPLE 3**
Address Masking—The C152 has two 8-bit address mask registers named AMSK0 and AMSK1. Bits in AMSK0 correspond to bits in ADR0 and bits in AMSK1 correspond to bits in ADR1. Placing a 1 into any bit position in AMSKn causes the corresponding bit in ADRn to be disregarded when searching for an address match.

To implement address masking:

\[ \text{AMSK0} = \text{n}nn\ldots\text{n}nn \]

and optionally:

\[ \text{AMSK1} = \text{n}nn\ldots\text{n}nn \]

where \(n = 1\) for a “don’t care address bit” or \(n = 0\) for a “do care address bit”

There are two main uses for the address masking capabilities of the C152. The first and simplest use is to mask off all address bits. In this mode the C152 will receive all messages. This type of reception is called “promiscuous” mode. The promiscuous mode could be used where all traffic would be monitored by a supervisor node to determine traffic patterns or to classify what information is being transferred between which nodes.

A second use of masking registers is to group various nodes together. Typically, stations are grouped together which have something in common, such as functions or location. Another term used when discussing group addresses is “multi-cast” addressing. Example #3 demonstrates how multi-cast addressing might be used.

Finally, to communicate with any station that has a printer, the address 00001001 would be sent and stations B, D, and E would receive the data. There are some limitations to using this type of scheme. Some of the more obvious are: the number of groupings is limited to the number of address bits minus 1, and it is not possible to address those stations that have a combination of attached peripherals, e.g., those stations with keyboards AND terminals. These problems can be solved using more elaborate addressing schemes.

HBA—Hardware Based Acknowledge (HBA) is a hardware implemented acknowledgment mechanism. The acknowledgement consists of a standalone preamble. An example of a preamble is shown in Figure 5. An acknowledgment will be returned by the receiver if:

- no hardware detectable errors are found in the frame
- the address is an individual address (LSB = 0)
- the transmitter is enabled (TEN = 1)
- HBA is set

An originating transmitter will expect and accept the acknowledgment if:

- HBA is set
- the receiver is enabled (GREN = 1)
- the address sent out was an individual address (LSB = 0)

If a partial or corrupted preamble is received or the preamble is not completed within the interframe space, the NOACK bit is set by the station that originally initiated transmission. HBA is a user selectable option which must be enabled after a reset.

The HBA method informs the original transmitter that a packet was received with no detected errors which saves the overhead and time that would normally be required to send a software generated acknowledgment for a valid reception. Some functions that other acknowledgment schemes implement yet are not encompassed when using HBA with a C152 is to identify packets which are out of sequence or frames which are of a wrong type.

To enable HBA:

\[ \text{RSTAT} = \text{XXXXXX}1 \]

INITIALIZATION—PROTOCOL INDEPENDENT

Discussion so far has centered on those elements of initialization which will vary according to the protocol being implemented. As such, the protocol in many cases will dictate what values to use for initialization. In addition, there are some parameters set during initialization that will remain the same regardless of which protocol is being implemented. There are also some parameters which may vary for reasons other than which protocol is being used. These parameters are grouped together to form the protocol independent initialization functions. The following sections cover these elements of initialization. The discussion of initialization parameters is complete when the text covering “Starting, Maintaining, and Ending Transmissions” begins.

CLEARING COLLISION COUNTER—A transmission collision detect counter (TCDCNT) keeps track of the number of collisions that have occurred. It does this by shifting a 1 into the LSB for each collision that occurs during transmission of the preamble. When TCDCNT overflows, the C152 stops transmitting and sets TCDT. Setting TCDT signals that too many collisions have occurred and can cause an interrupt. TCDT also is set if a collision occurs after the GSC has accessed TFIFO. During normal transmission, TCDCNT can be read by user software to determine the number of collisions, if any, that have occurred. Before starting the second and subsequent transmissions, it is possible that TCDCNT already has bits shifted in from a previous transmission. This would cause TCDCNT to over-
flow prematurely. In order to preserve the full bandwidth of 8 retransmissions, TCDCNT must be cleared prior to beginning any new transmission.

To clear the collision counter:
TCDCNT = 0

CONTROL OF THE GSC—“Control of the GSC” specifies how bytes are loaded into the transmitter (TFIFO) and unloaded from the receiver (RFIFO). A user has the choice of moving data to or from the GSC under control of either user software or the DMA channels.

CPU Control—CPU control is the simplest method of servicing the GSC and allows the most control. The major drawback to CPU control is that a significant amount of time is spent moving data from the source to the destination, incrementing pointers and counters, checking flags, and determining when the end of data occurs. In addition, how the GSC interrupts function differs from when the GSC is under CPU control than when the GSC is under DMA control. Under CPU control, valid GSC interrupts occur when either RFNE (Receive Fifo Not Empty) or TFNF (Transmit Fifo Not Full) are set. The transmit error and most of the receive error interrupts still function the same regardless of which type of control is used on the GSC. The only difference in how receive error interrupts operate is that the UR (UnderRun) bit for the receiver is operational when the GSC is under DMA control. UR is disabled when under CPU control.

DMA Control—DMA control relieves the CPU of much of the overhead associated with serving the GSC and allows faster baud rates. However, the reader must realize that more details about a “yet to be transmitted packet” must be known to properly initialize the DMA channels prior to starting a transmission. In some situations, especially at high baud rates, the user must take into account DMA cycles that occur asynchronously and without any user control or knowledge. This could possibly disrupt other time critical tasks the C152 is performing. There may be no indication to a user that other ongoing tasks are being interrupted by DMA cycles taking over the bus and momentarily stopping CPU action.

When the DMA is used to service the GSC, the DMA channels will also need to be initialized and the GSC interrupts configured to operate in DMA mode. The main advantages of using DMA control is time saved and interrupts occur only when there is an error or when the GSC operation (receive or transmit) is done. This removes the necessity of continuously polling RDN and TDN bits to determine when a GSC operation is complete.

One of the most important facts to remember when deciding how to service the GSC is that unless the GSC baud rate is relatively low compared to the CPU oscillator frequency, the only method that can keep up with the receiver or transmitter is DMA control. As a rule of thumb, if a user is willing to use 100% of available bandwidth of the C152 and no other interrupts are enabled besides the GSC, the maximum baud rate works out to be approximately 4.5% of the oscillator frequency. This is based on a 9 instruction cycle interrupt latency, moving a byte of data, return from interrupt and executing one more instruction before the next GSC byte is transmitted or received. At an oscillator frequency of 16 MHz, this works out to 720K bits per second. There are many steps a user could take to increase the baud rate when the GSC is under CPU control as this scenario is only a simple situation using worst case assumptions. Taking into account the amount of time available for the CPU to service the GSC as more tasks are required by the service routines or the CPU would further lower the maximum baud rate. For instance, if a user intended that GSC support only took 10% of available CPU time, this would reduce the effective baud rate by a factor of ten, making the maximum bit rate 72K. This 10% figure is an average over the period it takes to complete a frame. Situations might arise such that spurious GSC demand cycles would require much more than 10% of available time for short intervals.

INITIALIZING DMA—Since CSMA/CD is selected, it is by definition half-duplex. In half-duplex mode, only one DMA channel is needed to service both transmitter and receiver. However, it is simpler and easier to explain if both DMA channels are used. The following text is written under an assumption that both DMA channels will be used to service the GSC. Regardless of whether the DMA channel is servicing the receiver or transmitter, the DMA DONE interrupt generally should not be enabled. Also, the DMA bit in TSTAT must always be set. The GSC valid transmit and valid receive interrupts occur when RDN or TDN is set. This also eliminates a need to poll RDN or TDN to determine when a reception or transmission has ended, as is necessary when the GSC is under CPU control.

The DMA channel servicing the transmitter must have:
Destination Address = TFIFO (085H)
Increment Destination Address (IDA) = 0
Destination Address Space (DAS) = 1
Demand Mode (DM) = 1
Transfer Mode (TM) = 0

The source of data can be SFR space, internal RAM or external RAM. The byte count must be equal to the number of bytes to be transmitted, as this determines when a packet ends. TEN should be set before the DMA GO bit. It takes one bit time after TEN is set before the transmitter is enabled. The transmit valid
interrupt should be enabled after TEN is set. Since CSMA/CD is half duplex, it doesn't matter which DMA channel services the receiver or transmitter, as only one DMA channel will be active at any time.

The DMA channel servicing the receiver must have:
Source Address = RFIFO (0F4H)
ISA = 0
SAS = 1
DM = 1
TM = 0

The destination for data can be SFR space, internal RAM or external RAM. The byte count must be equal to or greater than the number of bytes to be received. Setting the byte count to 0FFFFFFH (64K) is one way of covering all packet lengths. GREN should be set after the DMA GO bit. The receive valid interrupt should be enabled after GREN is set. It takes one bit time after GREN is set before the receiver is enabled and for the error bits and RDN to be cleared. Before GREN is set, the user software should ensure that the RFIFO is cleared. Setting GREN does not clear the receive FIFO as stated in the hardware description.

INITIALIZING COUNTERS AND POINTERS: Whether using DMA or CPU control, pointers will be required to load the correct bytes for the transmitter and to store received bytes in their proper location. Counters are required when the GSC is under DMA control in order to keep the DMA channel active during the reception of an entire frame and to identify when a transmitted frame is to be ended. Counters are optional if the CPU is used to service the GSC, although its usefulness might be questioned.

When the GSC is under DMA control, the data pointers used are destination address registers (DARLn and DARHn) for the DMA channel responsible for the receiver and source address registers (SARLn and SARHn) for the DMA channel servicing the transmitter. The counters used are byte count registers (BCRLn and BCRHn) for the appropriate DMA channel.

The byte count for the transmitting DMA channel must be known and loaded prior to beginning actual transmission. Transmission begins when TEN and GO are set. The reason the byte count must be known prior to transmission is that when the counter reaches 0, the DMA stops loading data into TFIFO, and once TFIFO is emptied the GSC assumes a transmitted packet is complete. For the receiver the byte count can be set to the frame length if known prior to starting reception or the byte count can be set to a maximum frame packet length that will ever be received. Another alternative is to set the byte count equal to 0FFFFFFH. This option may be chosen if the length of received packets are totally unknown. If 0FFFFFFH is used, the user must make sure that there is some method to accommodate this many bytes. If maximum buffer size is a limiting factor, then that would be used.

When the GSC is under CPU control, internal RAM is typically used for pointers and counters. These pointers and counters would be updated by software for each byte that is received or transmitted. An interrupt is generated as long as there is at least one byte in the receive FIFO. An interrupt is also generated as long as there is room for one byte in the transmit FIFO. It is in the interrupt service routine that counters and pointers are updated and data is transferred to or from the GSC FIFOs. One advantage of CPU control is that the length of received or transmitted packets need not be known prior to the start of GSC activities. When the GSC is under CPU control, user software determines when a transmission has ended. For moving targets, CPU control allows the user software to determine where to store received data at the time it is transferred to RFIFO.

So far only initialization of the GSC and DMA has been explained. In order to use the GSC, the receiver, transmitter, and associated interrupts need to be enabled. These are covered in the following section.

ENABLING RECEIVER AND RECEIVER INTERRUPTS—There are two receiver interrupt enable bits, EGSRV (Receive Valid) and EGSRE (Receive Error) and one bit to enable the receiver (GREN). The interrupts should always be enabled whenever the receiver is enabled. Once this is done, a user can wait for interrupts to occur and then service the GSC receiver. The conditions which will cause the CPU to vector to GSC receiver interrupt service routines are described in the 8-Bit Embedded Controller Handbook.

In most CSMA/CD applications, GSC receivers will be enabled all the time once the C152 has been initialized. The only time the receiver will not be enabled is when a reception is completed or a receive error occurs. When this happens, the GSC receiver hardware clears GREN, which disables the receiver. The receiver must then be re-enabled by software before it is ready to accept a new frame. One way to do this when under DMA control is to set the receiver enable bit (GREN) in the receiver interrupt service routine. Similarly, the GSC receive interrupts should always be enabled and remain so except for the period of time that it takes to service an interrupt.

Once set, the GSC receiver interrupt enable bits always remain set unless cleared by user software. About the only valid reason for clearing the receiver interrupt enable bits is so that certain sections of code will not be disrupted by GSC activities. If the interrupts are disabled while the receiver is enabled, the amount of time the interrupts are disabled should not exceed 24 bit times. If the interrupts are disabled for a longer period of time, the receive FIFO may be over written.
It is a good practice to enable the GSC receiver interrupts prior to enabling the receiver when under CPU control. Another alternative is to clear the EA bit while enabling the GSC receiver and receiver interrupts. However, this could increase interrupt latency. If something like this is not done, a higher priority interrupt may alter the program flow immediately after the receiver is enabled and prior to enabling the interrupts. This in turn could cause the receiver to overflow. When the receiver is under DMA control the situation is different. First, the interrupts cannot be enabled before the receiver because if RDN is set from a previous reception, the receive valid service routine will be invoked but no reception has yet taken place. The correct sequence when under DMA control would be to set the DMA GO bit, enable the receiver, then enable the receiver interrupts. In this case the worst that could happen is a slow response to RDN getting set. Even this can be worked around by making receive valid the only high priority interrupt.

To enable the receiver interrupt enable bits and the receiver this sequence should be followed:

\[
\begin{align*}
IEN1 &= XXXXXXX1 \\
RSTAT &= XXXXXXX1X \\
\end{align*}
\]

or if under DMA control:

\[
\begin{align*}
DCONn &= XXXXXXX1 \\
RSTAT &= XXXXXXX1X \\
IEN1 &= XXXXXXX11 \\
\end{align*}
\]

ENABLING TRANSMITTER AND TRANSMIT INTERRUPTS—There are two transmit interrupt enable bits—EGSTV (Transmit Valid) and EGSTE (Transmit Error) and one transmitter enable bit—TEN (Transmitter Enable). The interrupts should always be enabled whenever the transmitter is enabled. Once this is done, a user can wait for interrupts to occur and then service the GSC transmitter. Conditions which will cause the CPU to vector to GSC transmit interrupt service routines are described in the 8-Bit Embedded Controller Handbook.

Compared with the receiver, opposite conditions exist concerning when the transmitter is operational and the sequence of enabling transmitter versus transmit interrupts. First, the transmitter and its interrupts are disabled all of the time except on those occasions when a transmission is desired. The user's application determines when a transmission is needed. Status of the message, how full a buffer is, or how long since the last message was sent are typical criteria used to judge when a transmission will be started.

When a transmission is complete, the interrupts and the transmitter should be disabled. This is particularly true for the transmit valid interrupt as TFIFO will most likely be empty and TFNF (Transmit FIFO Not Full) will be set. TFNF = 1 is the source of transmit valid interrupts when the GSC is serviced under CPU control.

The transmitter should be enabled before enabling the transmitter interrupts. If the GSC is under CPU control and the interrupts are enabled first, TFIFO may be loaded with data in response to TFNF being set. When TEN is set, data already loaded into TFIFO would be cleared. Consequently, data meant to be transmitted would be lost. If the GSC is under DMA control, it is possible that an interrupt would be generated in response to TDN being set from the previous transmission, yet no transmission has even started since the interrupts were enabled. If using the DMA channels to service the transmitter, TEN must be set before the GO bit for the DMA channel is set. If not, the DMA channels could load TFIFO with data, and when TEN is set that data would be lost.

The correct sequence to enable the transmitter and its interrupt enable bits is:

\[
\begin{align*}
\text{SETB TEN} \\
\text{SETB EGSTE} \\
\text{SETB EGSTV} \\
\end{align*}
\]

or if under DMA control:

\[
\begin{align*}
\text{SETB TEN} \\
\text{SETB EGSTE} \\
\text{SETB EGSTV} \\
\text{ORL DCONn, #01} \\
\end{align*}
\]

Once all initialization tasks shown so far are completed, reception and transmission may commence. The process of starting, maintaining, and ending transmissions or receptions is covered next.
STARTING, MAINTAINING, AND ENDING TRANSMISSIONS

Prior to starting a transmission, the user will need to set TEN. This enables the transmitter, resets TDN, clears all transmit error bits and sets up TFIFO as if it were empty (all bytes in TFIFO are lost) after a GSC bit clock occurs. Once TEN is set, actual transmission begins when a byte is loaded into TFIFO. Figure 9 is a block diagram of the GSC transmitter and shows how it functions. Once a byte has entered TFIFO, transmission begins. The first step is for the GSC to determine if the link is idle and interframe space has expired. Actually, this occurs continuously, even when not transmitting, but transmit circuitry checks to make sure these conditions exist before transmitting. If these two conditions are not met, the C152 will wait until they are. Once interframe space has expired, DEN is forced low for one bit time prior to the GSC emitting a preamble and BOF. About the time the BOF is output, a byte from TFIFO is transferred to the shift register. As bits are shifted out this register, they pass by the CRC generator, which updates the current CRC value. Bits then enter the data encoder which forms them into Manchester coded waveforms and out TxD. If TFIFO is empty when the shift register goes to grab another byte, the GSC assumes it is the end of data. To complete a frame, bits in the CRC generator are passed through the data encoder and the EOF is appended. One part of the block diagram in Figure 9 is the transmit control sequencer. The transmit control sequencer's purpose is to determine which state the transmitter is in such as Idle, Preamble, Data, or CRC. To perform this function it has connections to all circuits in the transmitter. These connections are not shown in order to make the diagram easier to read.

If the transmitter is under CPU control the first byte is loaded with user software. TFIFO should be filled and counters and pointers updated before proceeding with any other tasks required by the CPU. There is room for up to three bytes in TFIFO. Before loading the first byte, users should examine TDN to ensure that any previous transmissions have completed. If TEN is set before the end of a transmission, that transmission is aborted without appending a CRC and EOF but the interframe space will still be enforced before starting again. A user can identify when TFIFO is full by examining TFNF (Transmit Fifo Not Full). TFNF will always remain at a logic 1 as long as there is room for at least one more byte in TFIFO. There is a one machine cycle latency from when a byte is loaded into TFIFO until TFNF is updated. Because of this latency, the status of TFNF should not be checked immediately following the instruction that loaded TFIFO but should be examined two or more instructions later. Whenever TFNF is set, an interrupt will be generated if EGSTV is set. In response to the interrupt, bytes should be loaded into TFIFO until TFNF is cleared and update any pointers or counters.

Once the user is through with transmitting bytes for the current frame, the GSC transmit valid interrupt (EGSTV) should be disabled. This is to prevent the program flow from being interrupted by unnecessary GSC demands as TFNF will remain set all the time. The GSC transmit error interrupt (EGSTE) must remain enabled as transmit errors can still occur. While under CPU control there is no interrupt associated with transmit done (TDN) so a user must periodically poll this bit to determine when actual transmission is complete. After the last byte in TFIFO is transmitted there is a delay until TDN is set. This delay will be equal to the CRC length plus approximately 1.5 bit times for the EOF. The CRC is appended after the end of data by GSC hardware.
To start a transmission when the GSC is under DMA control, users should first enable the transmitter by setting TEN, then set the GO bit for the appropriate DMA channel. Before the GO bit is set users must initialize the GSC and DMA. Thereafter, the DMA loads the first byte that begins actual transmission and keeps the transmit FIFO full until the end of transmission. In this case, transmission ends when the byte count reaches 0, which means the length of the message to be transmitted must be known before transmission begins.

The DMA channel examines TFNF to determine when the transmitter needs servicing. When a byte is transferred into TFIFO, the DMA channel takes control of the internal bus and the CPU is held off for one machine cycle. This is the only overhead associated with the actual transmission when under DMA control. This is significantly less than the overhead associated with each byte that must be loaded by software when the GSC is under CPU control. When the DMA is servicing the transmitter, at least one machine cycle occurs between each DMA load. This prevents the DMA from hogging the internal bus when servicing the transmitter. It takes five machine cycles to load three bytes to initially fill TFIFO. When transmission ends, TDN will be set and when the GSC is under DMA control it is the setting of TDN that begins the GSC interrupt service routine.

The discussion so far assumes there are no errors during transmission of a frame. However, in CSMA/CD there is always a possibility of an error occurring and part of maintaining transmission is servicing those errors. In the C152 when an error is detected an error bit is set. At the same time the error bit is set, TEN is cleared which disables the transmitter. Types of errors that can occur are: collision detection errors (TCDT), no acknowledgement errors (NOACK) (if HBA is enabled), and underrun errors (UR) (if the DMA channels are used to service the transmitter). After setting the error bit, the C152 jumps to the transmit error vector if EGSTE (Transmit Error enable) is set. Depending on the protocol implemented, a user may wish to take some specific response to an error but in almost all cases the transmitter will be re-enabled and the same data retransmitted. This requires that counters and pointers be initialized, the transmitter enabled, and TFIFO filled. Another frequent action taken is to log the type of error for later analysis or to keep track of specific trends. Once transmission is restarted, the same flow is followed as before, as if no error occurred.

**STARTING, MAINTAINING, AND ENDING RECEIPTIONS**

In most applications, the receiver is always enabled and reception begins when the first byte is loaded into RFIFO. Figure 10 shows a block diagram of the receiver.

As indicated in Figure 10, before the first byte is loaded into RFIFO, the address is checked for a matching address assigned by ADRn. A user can disable address recognition by writing all 1s to the address mask registers, AMSKn. In this mode all frames with a valid BOF will be received. When the first byte is loaded into RFIFO, RFNE is set. If the address does match, there is a delay of about 24 or 40 bit times from reception of the first bit until a byte is loaded into RFIFO depending on which CRC is chosen. This is due to CRC strip circuitry and the bits required to fill up the shift register.

![Figure 10. Receiver Block Diagram](image-url)
When the GSC is being serviced by the CPU, an interrupt is generated when RFNE is set and if EGSRV is enabled. The user typically responds to an interrupt by removing one byte from RFIFO and storing it somewhere else. The user should check RFNE before leaving the interrupt service routine to see if more than one byte was loaded in to RFIFO. While under CPU control, there is no interrupt generated when reception is complete although receive done (RDN) is set. When RDN is set, the receiver is disabled and user software has to re-enable it. To determine when a frame has ended, the user must periodically poll RDN. After a frame has ended, the user will normally reinitialize pointers, reset counters, and enable the receiver. RDN will not be set when the last byte is transferred to RFIFO because the EOF will not be recognized yet. It takes approximately 1.7 bit times of link inactivity for the EOF to be recognized.

When the GSC is controlled by the DMA channels an interrupt is generated when RDN is set for a valid reception. At this point all a user needs to do is to set the source address registers, set the byte count, set the GO bit, and enable the receiver. Whenever the GSC receiver is being serviced by the DMA channels, the GO bit should be set before the receiver enable bit, GREN. This is to ensure that the DMA channel is active whenever the receiver is enabled. If the receiver is enabled before the DMA channel, it is possible that an interrupt would alter the program flow. An interrupt could delay setting the GO bit so that data is received while the DMA channel is prevented from servicing the GSC. Consequently, an overrun error occurs.

For the GSC receiver, as in the transmitter, an error is always possible. Conditions that set the error bits are the same regardless of how the receiver is being serviced. Possible errors are: receiver collision (RCABT), CRC error (CRCCE), overrun (OVR), and alignment error (AE).

The only type of error that user software can take actions to prevent is an overrun error. In this case, when an overrun error occurs it is because the receiver could not be serviced fast enough. Under DMA control, the only way this could happen is if the other DMA channel prevented servicing the GSC by the DMA or the user cleared the GO bit. Solutions to these problems are to turn off the second DMA channel when receiving and not mess around with the GO bit during reception. To determine if the GSC is receiving a packet, the byte count of the appropriate DMA channel can be examined. If the GSC is under CPU control and an overrun occurs it is because there are too many other tasks the CPU is doing or the baud rate is just too high for the CPU to keep up. A solution to this problem is to either cut back on the number of tasks the CPU must perform while a packet is being received or to switch to DMA control of the GSC.

In all other cases, about all the CI52 can do when a receive error occurs is to log the type of error, discard the data already received, and to re-enable the receiver for the next packet. These actions would also be taken for an overrun error.

**SUMMARY**

Hopefully, this application note has given the reader some insight on how to set up the GSC parameters, how to transmit or receive a packet, and how to respond to error conditions that may arise. The process of obtaining data for transmission or what to do with data received has been left open as much as possible as these vary widely from application to application. In some cases, all the data will be managed by another, more powerful processor. In this situation, the user will have to implement another interface between the main processor and the CI52.

Although the whole process of using the CI52 may at first, seem confusing and complicated, breaking down this process into steps may make utilizing the CI52 much simpler. One suggestion of the steps to follow is:

1) **INITIALIZATION**
   - A) Baud rate
   - B) Preamble
   - C) Backoff
   - D) CRC
   - E) Interframe space
   - F) Jamming signal
   - G) Slot time
   - H) Addressing
   - I) Acknowledgment
   - J) Clearing the collision counter
   - K) Controlling the GSC
   - L) DMA initialization (if used)
   - M) Counter and pointer setup
   - N) Enabling the GSC
   - O) Enabling the interrupts

2) **TRANSMITTING/RECEIVING PACKETS**
   - A) Starting transmission/reception
   - B) Maintaining GSC operations
   - C) Ending transmission/reception
   - D) Responding to errors

These steps can be used as a checklist to ensure that the minimum set of functions have been implemented that will allow the GSC to be used in almost any application. The list also demonstrates that the bulk of the tasks the user must implement is in initializing the GSC. Once initialization is accomplished, there is comparatively little work left to implement an application.
APPENDIX A
SOFTWARE EXAMPLE

The following example demonstrates how the DMA can be used to service the GSC in a specific environment. Figure 11 shows a diagram of the hardware used. As shown, the UART is used as a source and destination for data transferred by the GSC. Also shown in Figure 11 are some DIP switches. These DIP switches determine source and destination addresses. The switches are read only once after a reset. The hardware environment is shown for informational purposes only and is not necessarily a real application that would be implemented by a user. Even so, with some minor changes, similar circuits might be used, requiring corresponding changes to be made in the software.

This program has been written with the assumption that a terminal will be connected to the UART. As such, only ASCII data can be transferred and each block of data is delineated by a carriage return (0DH) and line feed (0AH). As data is received by the UART it is stored in one of four rotating buffers. This data will later be transmitted by the GSC to other C152s. Data received by the GSC is stored in one of four different rotating buffers. This data will be transmitted by the UART to a terminal. 1K of external data RAM is connected to the C152 to serve as storage buffers. Consequently, each buffer is one-eighth of available external RAM, or 128 bytes. This provides up to one line of 120 characters for each buffer. Also, each buffer will store additional information such as destination address, source address, and message length. When a line of characters is complete, a flag will be set to signify to the GSC that that buffer is to be transmitted. Conversely, when a packet received by the GSC is complete, a flag is set to identify that buffer is to be output through the UART to a terminal. Whenever access to one buffer is complete, the software manipulates pointers so the next buffer is used. If all 4 buffers are full, data for that type of buffer is no longer accepted until another buffer is available.

Note that this program uses both DMA channels, one for the receiver and one for the transmitter on the GSC. A program could have been written using only one DMA channel. Using both channels has made the program much simpler and shortened the time it takes to change from transmitting to receiving.
Figure 11. Hardware Environment for Software Example
### LOC OBJ . LINE SOURCE

<table>
<thead>
<tr>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
<td>@REF</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>@NLIST</td>
</tr>
<tr>
<td>000C</td>
<td>165</td>
<td>GSC_BAUD_RATE EQU 00</td>
</tr>
<tr>
<td></td>
<td>166</td>
<td>GSC_BAUD_RATE EQU 0FCH</td>
</tr>
<tr>
<td></td>
<td>167</td>
<td>LSC_BAUD_RATE EQU 0FCH</td>
</tr>
<tr>
<td></td>
<td>168</td>
<td>LSC_BAUD_RATE EQU 0FCH</td>
</tr>
<tr>
<td></td>
<td>169</td>
<td>LFST_PERIOD EQU 20</td>
</tr>
<tr>
<td>0014</td>
<td>170</td>
<td>BUF1A_START_ADDR EQU 003H</td>
</tr>
<tr>
<td></td>
<td>171</td>
<td>BUF1A_START_ADDR EQU 003H</td>
</tr>
<tr>
<td></td>
<td>172</td>
<td>BUF1A_START_ADDR EQU 003H</td>
</tr>
<tr>
<td></td>
<td>173</td>
<td>BUF1A_START_ADDR EQU 003H</td>
</tr>
<tr>
<td>0083</td>
<td>174</td>
<td>BUF1B_START_ADDR EQU 083H</td>
</tr>
<tr>
<td></td>
<td>175</td>
<td>BUF1B_START_ADDR EQU 083H</td>
</tr>
<tr>
<td></td>
<td>176</td>
<td>BUF1B_START_ADDR EQU 083H</td>
</tr>
<tr>
<td></td>
<td>177</td>
<td>BUF1B_START_ADDR EQU 083H</td>
</tr>
<tr>
<td>0103</td>
<td>178</td>
<td>BUF1C_START_ADDR EQU 103H</td>
</tr>
<tr>
<td></td>
<td>179</td>
<td>BUF1C_START_ADDR EQU 103H</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>BUF1C_START_ADDR EQU 103H</td>
</tr>
<tr>
<td></td>
<td>181</td>
<td>BUF1C_START_ADDR EQU 103H</td>
</tr>
<tr>
<td>0183</td>
<td>182</td>
<td>BUF1D_START_ADDR EQU 183H</td>
</tr>
<tr>
<td></td>
<td>183</td>
<td>BUF1D_START_ADDR EQU 183H</td>
</tr>
<tr>
<td></td>
<td>184</td>
<td>BUF1D_START_ADDR EQU 183H</td>
</tr>
<tr>
<td></td>
<td>185</td>
<td>BUF1D_START_ADDR EQU 183H</td>
</tr>
<tr>
<td>0201</td>
<td>186</td>
<td>BUF2A_START_ADDR EQU 201H</td>
</tr>
<tr>
<td></td>
<td>187</td>
<td>BUF2A_START_ADDR EQU 201H</td>
</tr>
<tr>
<td></td>
<td>188</td>
<td>BUF2A_START_ADDR EQU 201H</td>
</tr>
<tr>
<td></td>
<td>189</td>
<td>BUF2A_START_ADDR EQU 201H</td>
</tr>
<tr>
<td>0281</td>
<td>190</td>
<td>BUF2B_START_ADDR EQU 281H</td>
</tr>
<tr>
<td></td>
<td>191</td>
<td>BUF2B_START_ADDR EQU 281H</td>
</tr>
<tr>
<td></td>
<td>192</td>
<td>BUF2B_START_ADDR EQU 281H</td>
</tr>
<tr>
<td></td>
<td>193</td>
<td>BUF2B_START_ADDR EQU 281H</td>
</tr>
<tr>
<td>0301</td>
<td>194</td>
<td>BUF2C_START_ADDR EQU 301H</td>
</tr>
<tr>
<td></td>
<td>195</td>
<td>BUF2C_START_ADDR EQU 301H</td>
</tr>
<tr>
<td></td>
<td>196</td>
<td>BUF2C_START_ADDR EQU 301H</td>
</tr>
<tr>
<td></td>
<td>197</td>
<td>BUF2C_START_ADDR EQU 301H</td>
</tr>
<tr>
<td>0381</td>
<td>198</td>
<td>BUF2D_START_ADDR EQU 381H</td>
</tr>
<tr>
<td></td>
<td>199</td>
<td>BUF2D_START_ADDR EQU 381H</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>BUF2D_START_ADDR EQU 381H</td>
</tr>
<tr>
<td></td>
<td>201</td>
<td>BUF2D_START_ADDR EQU 381H</td>
</tr>
<tr>
<td>0080</td>
<td>202</td>
<td>STACK_OFFSET EQU 80H</td>
</tr>
<tr>
<td></td>
<td>203</td>
<td>CR EQU 0DH</td>
</tr>
<tr>
<td></td>
<td>204</td>
<td>CR EQU 0DH</td>
</tr>
<tr>
<td></td>
<td>205</td>
<td>CR EQU 0DH</td>
</tr>
<tr>
<td></td>
<td>206</td>
<td>LINE_FEED EQU 0AH</td>
</tr>
<tr>
<td></td>
<td>207</td>
<td>LINE_FEED EQU 0AH</td>
</tr>
<tr>
<td></td>
<td>208</td>
<td>LINE_FEED EQU 0AH</td>
</tr>
<tr>
<td>0000</td>
<td>209</td>
<td>ERROR_POINTER EQU RO</td>
</tr>
<tr>
<td></td>
<td>210</td>
<td>ERROR_POINTER EQU RO</td>
</tr>
<tr>
<td></td>
<td>211</td>
<td>ERROR_POINTER EQU RO</td>
</tr>
<tr>
<td></td>
<td>212</td>
<td>ERROR_POINTER EQU RO</td>
</tr>
</tbody>
</table>

**Notes:**
- GSC baud rate = 1.5MBps
- LSC baud rate = 9.6K baud at 14.7456 MHz
- Number of bit times separating frames
- Buffer 1A's starting address for storing data (0 = # of bytes, 1 = dest addr, 2 = src addr)
- Buffer 1B's starting address for storing data (0B0H = # of bytes, 0B1 = dest addr, 0B2 = src addr)
- Buffer 1C's starting address for storing data (100H = # of bytes, 101 = dest addr, 102 = src addr)
- Buffer 1D's starting address for storing data (180H = # of bytes, 181 = dest addr, 182 = src addr)
- Buffer 2A's starting address for storing data (200H = # of bytes)
- Buffer 2B's starting address for storing data (280H = # of bytes)
- Buffer 2C's starting address for storing data (300H = # of bytes)
- Buffer 2D's starting address for storing data (380H = # of bytes)
- Stack start at upper 128 bytes
- @REF
- @NLIST
- GSC baud rate
- LSC baud rate
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>007B</td>
<td></td>
<td>213</td>
<td>MAX_LENGTH EQU 120</td>
</tr>
<tr>
<td>214</td>
<td></td>
<td></td>
<td>maximum length a (received) packet</td>
</tr>
<tr>
<td>215</td>
<td></td>
<td></td>
<td>can be - must always be less than</td>
</tr>
<tr>
<td>216</td>
<td></td>
<td></td>
<td>255. my H/W limitation is 12B</td>
</tr>
<tr>
<td>217</td>
<td></td>
<td></td>
<td>;----------------------------------------------------------</td>
</tr>
<tr>
<td>00F</td>
<td></td>
<td>219</td>
<td>UR_COUNTER DATA OFFH</td>
</tr>
<tr>
<td>220</td>
<td></td>
<td></td>
<td>RAM locations OFAH to OFFH are</td>
</tr>
<tr>
<td>221</td>
<td></td>
<td></td>
<td>used to keep a log of the # of</td>
</tr>
<tr>
<td>222</td>
<td></td>
<td></td>
<td>UR errors (only transmit error)</td>
</tr>
<tr>
<td>00F9</td>
<td></td>
<td>223</td>
<td>OVR_COUNTER DATA (UR_COUNTER) - 6</td>
</tr>
<tr>
<td>224</td>
<td></td>
<td></td>
<td>RAM locations OFAH to OF9H keep</td>
</tr>
<tr>
<td>225</td>
<td></td>
<td></td>
<td>a log of the # of overrun errors</td>
</tr>
<tr>
<td>00F3</td>
<td></td>
<td>226</td>
<td>RCABT_COUNTER DATA (OVR_COUNTER) - 6</td>
</tr>
<tr>
<td>227</td>
<td></td>
<td></td>
<td>RAM locations OEH to OF3H keep</td>
</tr>
<tr>
<td>228</td>
<td></td>
<td></td>
<td>a log of the # of abort errors</td>
</tr>
<tr>
<td>00ED</td>
<td></td>
<td>229</td>
<td>AE_COUNTER DATA (RCABT_COUNTER) - 6</td>
</tr>
<tr>
<td>230</td>
<td></td>
<td></td>
<td>RAM locations OECDH to OEDH keep</td>
</tr>
<tr>
<td>231</td>
<td></td>
<td></td>
<td>a log of the # of alignment errors</td>
</tr>
<tr>
<td>00E7</td>
<td></td>
<td>232</td>
<td>CRC_COUNTER DATA (AE_COUNTER) - 6</td>
</tr>
<tr>
<td>233</td>
<td></td>
<td></td>
<td>RAM locations OECDH to OED7H keep</td>
</tr>
<tr>
<td>234</td>
<td></td>
<td></td>
<td>a log of the # of CRC errors</td>
</tr>
<tr>
<td>00E1</td>
<td></td>
<td>235</td>
<td>LONG_COUNTER DATA (CRC_COUNTER) - 6</td>
</tr>
<tr>
<td>236</td>
<td></td>
<td></td>
<td>RAM locations OECDH to OEDCH keep</td>
</tr>
<tr>
<td>237</td>
<td></td>
<td></td>
<td>a log of the # of received</td>
</tr>
<tr>
<td>238</td>
<td></td>
<td></td>
<td>packets that are too long</td>
</tr>
<tr>
<td>00DB</td>
<td></td>
<td>239</td>
<td>TCDT_COUNTER DATA (LONG_COUNTER) - 6</td>
</tr>
<tr>
<td>240</td>
<td></td>
<td></td>
<td>RAM locations ODBH to ODBH keep</td>
</tr>
<tr>
<td>241</td>
<td></td>
<td></td>
<td>a log of the # of TCDT errors</td>
</tr>
<tr>
<td>00D5</td>
<td></td>
<td>242</td>
<td>NOACK_COUNTER DATA (TCDT_COUNTER) - 6</td>
</tr>
<tr>
<td>243</td>
<td></td>
<td></td>
<td>RAM locations ODBH to ODBH keep</td>
</tr>
<tr>
<td>244</td>
<td></td>
<td></td>
<td>a log of the # of NOACK errors</td>
</tr>
<tr>
<td>00CF</td>
<td></td>
<td>245</td>
<td>NEXT_LOCATION DATA (NOACK_COUNTER) - 6</td>
</tr>
<tr>
<td>246</td>
<td></td>
<td></td>
<td>reserve 6 bytes for NOACK counter</td>
</tr>
<tr>
<td>247</td>
<td></td>
<td></td>
<td>;----------------------------------------------------------</td>
</tr>
<tr>
<td>007F</td>
<td></td>
<td>249</td>
<td>IN_BYTE_COUNT DATA 7FH</td>
</tr>
<tr>
<td>250</td>
<td></td>
<td></td>
<td>number of bytes LSC received which</td>
</tr>
<tr>
<td>251</td>
<td></td>
<td></td>
<td>determines # of bytes for GSC to</td>
</tr>
<tr>
<td>252</td>
<td></td>
<td></td>
<td>transmit</td>
</tr>
<tr>
<td>007E</td>
<td></td>
<td>253</td>
<td>OUT_BYTE_COUNT DATA (IN_BYTE_COUNT) - 1</td>
</tr>
<tr>
<td>254</td>
<td></td>
<td></td>
<td>number of bytes GSC received which</td>
</tr>
<tr>
<td>255</td>
<td></td>
<td></td>
<td>determines # of bytes for LSC to</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td>transmit</td>
</tr>
<tr>
<td>007D</td>
<td></td>
<td>257</td>
<td>GSC_DEST_ADDR DATA (OUT_BYTE_COUNT) - 1</td>
</tr>
<tr>
<td>258</td>
<td></td>
<td></td>
<td>destination address read from</td>
</tr>
<tr>
<td>259</td>
<td></td>
<td></td>
<td>DIP switches (loaded on RESET)</td>
</tr>
<tr>
<td>007C</td>
<td></td>
<td>260</td>
<td>GSC_SRC_ADDR DATA (GSC_DEST_ADDR) - 1</td>
</tr>
<tr>
<td>261</td>
<td></td>
<td></td>
<td>source address read from DIP</td>
</tr>
<tr>
<td>262</td>
<td></td>
<td></td>
<td>switches (loaded on RESET)</td>
</tr>
<tr>
<td>007B</td>
<td></td>
<td>263</td>
<td>LSC_INPUT_LOW DATA (GSC_SRC_ADDR) - 1</td>
</tr>
<tr>
<td>264</td>
<td></td>
<td></td>
<td>contains the address where the</td>
</tr>
<tr>
<td>265</td>
<td></td>
<td></td>
<td>next LSC received byte will be</td>
</tr>
<tr>
<td>266</td>
<td></td>
<td></td>
<td>stored at</td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ</td>
<td>LINE</td>
<td>SOURCE</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>0079</td>
<td>268</td>
<td>GSC_INPUT_LOW DATA</td>
<td>(LSC_INPUT_HIGH) - 1</td>
</tr>
<tr>
<td>0078</td>
<td>269</td>
<td>GSC_INPUT_HIGH DATA</td>
<td>(GSC_INPUT_LOW) - 1</td>
</tr>
<tr>
<td>270</td>
<td>270</td>
<td></td>
<td>contains the address where the next GSC received byte will be stored</td>
</tr>
<tr>
<td>271</td>
<td>271</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0077</td>
<td>272</td>
<td>LSC_OUTPUT_LOW DATA</td>
<td>(GSC_INPUT_HIGH) - 1</td>
</tr>
<tr>
<td>0076</td>
<td>273</td>
<td>LSC_OUTPUT_HIGH DATA</td>
<td>(LSC_OUTPUT_LOW) - 1</td>
</tr>
<tr>
<td>274</td>
<td>274</td>
<td></td>
<td>contains the address where the next byte for the LSC to transmit is</td>
</tr>
<tr>
<td>275</td>
<td>275</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0075</td>
<td>276</td>
<td>LSC_OUT_COUNTER DATA</td>
<td>(LSC_OUTPUT_HIGH) - 1</td>
</tr>
<tr>
<td>277</td>
<td>277</td>
<td></td>
<td>contains the number of byte for the LSC to transmit</td>
</tr>
<tr>
<td>278</td>
<td>278</td>
<td></td>
<td></td>
</tr>
<tr>
<td>002F</td>
<td>279</td>
<td>BUFFER1_CONTROL DATA</td>
<td>2FH</td>
</tr>
<tr>
<td>280</td>
<td>280</td>
<td></td>
<td>byte that buffer 1 control bits are in</td>
</tr>
<tr>
<td>281</td>
<td>281</td>
<td></td>
<td></td>
</tr>
<tr>
<td>002E</td>
<td>282</td>
<td>BUFFER2_CONTROL DATA</td>
<td>2EH</td>
</tr>
<tr>
<td>283</td>
<td>283</td>
<td></td>
<td>byte that buffer 2 control bits are in</td>
</tr>
<tr>
<td>284</td>
<td>284</td>
<td></td>
<td></td>
</tr>
<tr>
<td>285</td>
<td>285</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0077</td>
<td>286</td>
<td>BUFD_ACTIVE BIT 7FH</td>
<td></td>
</tr>
<tr>
<td>287</td>
<td>287</td>
<td></td>
<td>indicator for when buffer 1D has data for GSC</td>
</tr>
<tr>
<td>288</td>
<td>288</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007E</td>
<td>289</td>
<td>BUF1C_ACTIVE BIT (BUFD_ACTIVE) - 1</td>
<td></td>
</tr>
<tr>
<td>290</td>
<td>290</td>
<td></td>
<td>indicator for when buffer 1C has data for GSC</td>
</tr>
<tr>
<td>291</td>
<td>291</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007D</td>
<td>292</td>
<td>BUF1B_ACTIVE BIT (BUF1C_ACTIVE) - 1</td>
<td></td>
</tr>
<tr>
<td>293</td>
<td>293</td>
<td></td>
<td>indicator for when buffer 1B has data for GSC</td>
</tr>
<tr>
<td>294</td>
<td>294</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007C</td>
<td>295</td>
<td>BUF1A_ACTIVE BIT (BUF1B_ACTIVE) - 1</td>
<td></td>
</tr>
<tr>
<td>296</td>
<td>296</td>
<td></td>
<td>indicator for when buffer 1A has data for GSC</td>
</tr>
<tr>
<td>297</td>
<td>297</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007B</td>
<td>298</td>
<td>GSC_OUT_MSB BIT (BUF1A_ACTIVE) - 1</td>
<td></td>
</tr>
<tr>
<td>299</td>
<td>299</td>
<td></td>
<td>second of two bits that identify which buffer is the current GSC output buffer</td>
</tr>
<tr>
<td>300</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007A</td>
<td>301</td>
<td>GSC_OUT_LSB BIT (GSC_OUT_MSB) - 1</td>
<td></td>
</tr>
<tr>
<td>302</td>
<td>302</td>
<td></td>
<td>first of two bits that identify which buffer is the current GSC output buffer</td>
</tr>
<tr>
<td>303</td>
<td>303</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0079</td>
<td>304</td>
<td>LSC_IN_MSB BIT (GSC_OUT_LSB) - 1</td>
<td></td>
</tr>
<tr>
<td>305</td>
<td>305</td>
<td></td>
<td>second of two bits that identify which buffer is the current LSC input buffer</td>
</tr>
<tr>
<td>306</td>
<td>306</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0078</td>
<td>307</td>
<td>LSC_IN_LSB BIT (LSC_IN_MSB) - 1</td>
<td></td>
</tr>
<tr>
<td>308</td>
<td>308</td>
<td></td>
<td>first of two bits that identify which buffer is the current LSC input buffer</td>
</tr>
<tr>
<td>309</td>
<td>309</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0077</td>
<td>310</td>
<td>BUF2A_ACTIVE BIT (LSC_IN_LSB) - 1</td>
<td></td>
</tr>
<tr>
<td>311</td>
<td>311</td>
<td></td>
<td>indicator for when buffer 2A has data for LSC</td>
</tr>
<tr>
<td>312</td>
<td>312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0076</td>
<td>313</td>
<td>BUF2B_ACTIVE BIT (BUF2A_ACTIVE) - 1</td>
<td></td>
</tr>
<tr>
<td>314</td>
<td>314</td>
<td></td>
<td>indicator for when buffer 2B has data for LSC</td>
</tr>
<tr>
<td>315</td>
<td>315</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0075</td>
<td>316</td>
<td>BUF2C_ACTIVE BIT (BUF2B_ACTIVE) - 1</td>
<td></td>
</tr>
<tr>
<td>317</td>
<td>317</td>
<td></td>
<td>indicator for when buffer 2C has</td>
</tr>
<tr>
<td>318</td>
<td>318</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LOC OBJ | LINE | SOURCE
---|---|---
0074 | BUFD_ACTIVE | BIT (BUF2D_ACTIVE) - 1 indicator for when buffer 2D has
0073 | GSC_IN_MSB | BIT (BUF2D_ACTIVE) - 1 indicator for when buffer 2D has
0072 | GSC_IN_LSB | BIT (GSC_IN_MSB) - 1 first of two bits that identify
0071 | LSC_OUT_MSB | BIT (GSC_IN_LSB) - 1 second of two bits that identify
0070 | LSC_OUT_LSB | BIT (LSC_OUT_MSB) - 1 first of two bits that identify
006F | FIRST_GSC_OUT | BIT (LSC_OUT_LSB) - 1 indicator for first GSC xmit
006E | LSC_ACTIVE | BIT (FIRST_GSC_OUT) -1 indicator that LSC is outputting

START:
ORG 0
..IMP 0 INITIALIZATION
ORG 23H
JMP LSC_SERVICE
ORG 2BH
GSC_REC_VALID:
..IMP GSC_VALID_REC
ORG 33H
GSC_REC_ERROR:
..IMP GSC_ERROR_REC
ORG 43H
GSC_XMIT_VALID:
..IMP GSC_VALID_XMIT
ORG 4BH
GSC_XMIT_ERROR:
..IMP GSC_ERROR_XMIT
ORG 53H
DMA1.Done:
..IMP DMA1_SERVICE
006F 0204E3 | JMP GSC_ERROR_XMIT
0053 0204D5 | JMP GSC_ERROR_XMIT
0053 0204D1C | JMP DMA1_SERVICE
LOC  OBJ  LINE  SOURCE
0100  378   ORG 100H  INITIALIZATION
0100  380   MOV SP, #STACK_OFFSET  set stack at user defined addr
0103  383   CALL ADDRESS_DETERMINATION  setup addressing (only done on
0106  385   CALL OSC_INIT  initialization for OSC
0109  387   CALL LSC_INIT  initialization for LSC
010C  388   CALL GENERIC_INIT  general initialization not dealing
0112  389   CALL INTERRUPT_ENABLE  enable interrupts
0115  390   JB BUF1A_ACTIVE.BUFFER1_START  ;see if buffer 1A has something
0118  393   JB BUF1B_ACTIVE.BUFFER1_START  ;to transmit out OSC
011B  395   JB BUF1C_ACTIVE.BUFFER1_START  ;to transmit out OSC
011E  397   JB BUF1D_ACTIVE.BUFFER1_START  ;to transmit out OSC
0121  399   JB BUF2A_ACTIVE.BUFFER2_START  ;see if buffer 2A has something
0124  401   JB BUF2B_ACTIVE.BUFFER2_START  ;to transmit out LSC
0127  403   JB BUF2C_ACTIVE.BUFFER2_START  ;to transmit out LSC
012A  405   JMP MAIN  
012C  408   CALL NEW_BUFFER1_OUT  ;this routine should start a
012F  410   JMP MAIN  
0131  413   CALL NEW_BUFFER2_OUT  ;this routine starts a transmission
MCS-51 MACRO ASSEMBLER

10/19/88 PAGE 6

LOC OBJ LINE SOURCE

433
0134 86DC 434
0200 759400 435
0203 758402 436
0206 75A414 437
0209 750400 438
020C D2DB 439
0211 759298 440
0214 75B2F4 441
0217 75F300 442
021A 75F278 443
021D 759369 444
0220 857C95 445
0223 757901 446
0226 757802 447
0229 857902 448
022C 857803 449
022F D2E9 450
0231 D26F 451
0233 22 452
0234 7580FC 453

JMP MAIN
ORG 200H
$INCLUDE (GSCINIT.SRC)

MOV BAUD. GSC_BAUD_RATE
MOV GSC_GMND. GSC_BAUD_RATE
MOV IFS. IFS_PERIOD
MOV IFSDCT. #0
SETB DMA
MOV DARLO. TFIFO
MOV DCONO. 10011000B
MOV SARLI. RFIFO
MOV BCRHI. #0
MOV DCONI. #1011001B
MOV GSC_INPUT_LOW . LOW (BUF2A_S1RT_ADDR)
MOV GSC_INPUT_HIGH . HIGH (BUF2A_S1RT_ADDR)
MOV DARLO. GSC_INPUT_LOW
MOV DARHI. GSC_INPUT_HIGH
MOV GSC_INPUT_LOW
MOV GSC_INPUT_HIGH
MOV DARHI. GSC_INPUT_LOW
MOV DARLI. GSC_INPUT_HIGH
MOV DARHI. GSC_INPUT_LOW
MOV DARLO. GSC_INPUT_HIGH
SETB GREN
SETB FIRST_GSC_OUT
RET

JMP MAIN
$INCLUDE (GSCINIT.SRC)

GSC_INIT:

MOV TH1. #LSC_BAUD_RATE

;out the LSC if one of the buffers is full
;init for CSMA/CD. 8-bit preamble.
;16-bit CRC. 8-bit addresses
;IIFS for period between frames
;clear collision counter
;init GSC interrupts for DMA
;DMA0 will service TFIFO
;DMA0 with SFR as dest. ext. RAM
;as source. serial port demand mode
;DMA1 will service RFIFO
;load DMA byte count with maximum message length
;init DMA0 with ext. RAM as dest.
;SFR as source. serial port demand
;mode, and set GO bit.

;init GSC input address storage
;init GSC input address storage
;init DMA destination address to match GSC input address storage
;enable receiver
;set indicator that first GSC xmit has not yet occurred
;setup timer1 to generate LSC baud
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0237</td>
<td>43B920</td>
<td>=1</td>
<td>488</td>
</tr>
</tbody>
</table>
| 023A | 53892F | =1 | 489 | ORL TMOD, #0010000D  
| 023D | 759850 | =1 | 490 | ANL TMOD, #0011111B  
| 0240 | D2BE | =1 | 491 | MOV SCON, #01010000  
| 0242 | 22 | =1 | 492 | SETB TR1  
| 0244 | 53901F | =1 | 493 | RET |
| 0246 | B5C07C | =1 | 494 | SOURCE DRL TMDD.IOO.IOB  
| 0249 | 399020 | =1 | 495 | ANL TMDD.IOO.IOB  
| 024C | B5C07D | =1 | 496 | MOV TMDD.IOO.IOB  
| 024E | 22 | =1 | 497 | RET |
| 0250 | D2CB | =1 | 498 | *INCLUDE (INITADDR SRC)  
| 0252 | D2C9 | =1 | 499 | ADDR_DETERMINATION  
| 0254 | D2AC | =1 | 500 |  
| 0256 | D2CC | =1 | 501 | MOV GSC_SRC_ADDR.P4  
| 0258 | D2AF | =1 | 502 | RET |
| 025A | 22 | =1 | 503 | *INCLUDE (ENAINT SRC)  
| 025E | 752F00 | =1 | 504 | INTERRUPT_ENABLE  
| 0261 | C26E | =1 | 505 | SETB EGSR  
| 0263 | 757B03 | =1 | 506 | SETB EGSRE  
| 0265 | 22 | =1 | 507 | SETB ES  
| 0266 | 22 | =1 | 508 | MOV GSC_DEST_ADDR.P4  
| 0267 | 22 | =1 | 509 | RET |
| 0269 | 22 | =1 | 510 | *INCLUDE (GENINIT.SRC)  
| 026B | 22 | =1 | 511 | GENERIC_INIT  
| 026D | 22 | =1 | 512 | SETB EDMA1  
| 026E | 22 | =1 | 513 | SETB EA  
| 0270 | 22 | =1 | 514 | RET |
| 0272 | 22 | =1 | 515 | *INCLUDE (GENINIT.SRC)  
| 0274 | 22 | =1 | 516 | GENERIC_INIT  
| 0276 | 22 | =1 | 517 | MOV BUFFER1_CONTROL, #0  
| 0278 | 22 | =1 | 518 | MOV BUFFER2_CONTROL, #0  
| 027A | 22 | =1 | 519 | CLR LSC_ACTIVE  
| 027C | 22 | =1 | 520 | MOV LSC_INPUT_LOW, #0LOW (BUF1A_START_ADDR)  

- Init timer1 as 8-bit auto-reload  
- Setup GSC as 8-bit UART and enable receiver  
- Start timer to generate baud rate  
- Select output 0 of '138  
- Read GSC receive address from DIP switch #1  
- Select output 1 of '138  
- Read GSC emit address from DIP switch #2  
- Enable GSC receive valid interrupt  
- Enable GSC receive error interrupt  
- Enable LSC interrupt  
- Enable DMA1 done interrupt  
- Enable interrupts  
- Insure all buffer 1 active bits  
- Insure all buffer 2 active bits  
- Insure LSC_ACTIVE = 0 before starting a reception  

270720-23
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0266</td>
<td>757A00</td>
<td>=1</td>
<td>MOV LSC_INPUT_HIGH:#HIGH (BUF1A_START_ADDR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; load address pointers with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; starting address of buffer 1A</td>
</tr>
<tr>
<td>0269</td>
<td>757F02</td>
<td>=1</td>
<td>MOV IN_BYTE_COUNT:#02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; byte count initialized to 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; because destination and source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; address will take first two bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; and counter is not incremented.</td>
</tr>
<tr>
<td>026C</td>
<td>78CF</td>
<td>=1</td>
<td>MOV RO,#NEXT_LOCATION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>=1</td>
</tr>
<tr>
<td>026E</td>
<td>0E</td>
<td>=1</td>
<td>COUNTER_CLEAR:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>INC RO</td>
</tr>
<tr>
<td>026F</td>
<td>7600</td>
<td>=1</td>
<td>MOV #RO,#0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; clear out error counter area</td>
</tr>
<tr>
<td>0271</td>
<td>8F83</td>
<td>=1</td>
<td>CJNE RO,#OFFH,COUNTER_CLEAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; loop until all counters = 0</td>
</tr>
<tr>
<td>0274</td>
<td>22</td>
<td>=1</td>
<td>RET</td>
</tr>
<tr>
<td>0275</td>
<td>D3</td>
<td>=1</td>
<td>564 ++1 #INCLUDE (CTRINC.SRC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>565 INC_INCREMENT_COUNTER:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>566</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>567 SETB C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; add 1 on first loop</td>
</tr>
<tr>
<td>0276</td>
<td>7F06</td>
<td>=1</td>
<td>MOV R7,#6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; # of bytes in each counter field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>570 INC_COUNT_LOOP:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>571</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>572</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>573 MOV A,#ERROR_POINTER_A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; get byte of counter</td>
</tr>
<tr>
<td>0278</td>
<td>1B</td>
<td>=1</td>
<td>574 ADDC A,#0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>575</td>
</tr>
<tr>
<td>027B</td>
<td>F6</td>
<td>=1</td>
<td>576 MOV #ERROR_POINTER_A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>577</td>
</tr>
<tr>
<td>027C</td>
<td>1B</td>
<td>=1</td>
<td>578 DEC ERROR_POINTER_A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>579</td>
</tr>
<tr>
<td>027D</td>
<td>DFF9</td>
<td>=1</td>
<td>580 DJNZ R7,INC_COUNT_LOOP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>581</td>
</tr>
<tr>
<td>027F</td>
<td>4001</td>
<td>=1</td>
<td>582 JC COUNTER_OVERFLOW</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; overflow if carry generated. This</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; was initially put in to stop the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; flow of the program if any of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; error counters overflowed with the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; expectation that the user would</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; modify the code to dump the error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; account contents and re-initialize the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; counter locations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>583</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>584</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>585</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>586</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>587</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>588</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>589</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>590</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>591</td>
</tr>
<tr>
<td>0281</td>
<td>22</td>
<td>=1</td>
<td>RET</td>
</tr>
<tr>
<td>0282</td>
<td>0B</td>
<td>=1</td>
<td>COUNTER_OVERFLOW:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>593</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>594</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>595</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>INCREMENT_POINTER_A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=1</td>
<td>; point to msb of counter field</td>
</tr>
</tbody>
</table>

APPNOT1
10/19/98
PAGE 8
LOC  OBJ  LINE  SOURCE
0283 76FF  =1 598  MOV @ERROR_POINTER,OFFH  and store OFFH
0285 08  =1 599  INC ERROR_POINTER  point to next byte of counter field
0286 76FF  =1 600  MOV @ERROR_POINTER,OFFH  and store OFFH
0288 08  =1 601  INC ERROR_POINTER  point to next byte of counter field
0289 76FF  =1 602  MOV @ERROR_POINTER,OFFH  and store OFFH
028B 08  =1 603  INC ERROR_POINTER  point to next byte of counter field
028C 76FF  =1 604  MOV @ERROR_POINTER,OFFH  and store OFFH
028E 08  =1 605  INC ERROR_POINTER  point to next byte of counter field
028F 76FF  =1 606  MOV @ERROR_POINTER,OFFH  and store OFFH
0291 08  =1 607  INC ERROR_POINTER  point to next byte of counter field
0292 76FF  =1 608  MOV @ERROR_POINTER,OFFH  and store OFFH
0294 80FF  =1 609  MOV @ERROR_POINTER,OFFH  and store OFFH
0296 08  =1 610  INC ERROR_POINTER  point to next byte of counter field
0297 76FF  =1 611  MOV @ERROR_POINTER,OFFH  and store OFFH
0299 08  =1 612  INC ERROR_POINTER  point to next byte of counter field
029A 76FF  =1 613  MOV @ERROR_POINTER,OFFH  and store OFFH
029C 08  =1 614  INC ERROR_POINTER  point to next byte of counter field
029D 76FF  =1 615  MOV @ERROR_POINTER,OFFH  and store OFFH
029F 08  =1 617  INC ERROR_POINTER  point to next byte of counter field
02A1 76FF  =1 618  MOV @ERROR_POINTER,OFFH  and store OFFH
02A3 08  =1 619  INC ERROR_POINTER  point to next byte of counter field
02A5 80FF  =1 620  JMP *  If the error counters overflow the
02A7 18  =1 621  program continues to loop at this
02A9 18  =1 622  location until H/W resets the device.
02A9 08  =1 623  INCLUDE (BUFIMGT_SRCI
02AA 08  =1 624  NEW_BUFFER1_IN:
02AB 08  =1 626  ;******************************************************************************
02AC 08  =1 627  ;This section uses a bit addressable control byte to determine which buffers
02AD 08  =1 628  ;are active (contains data for GSC to output), the last buffer used by the LSC
02AE 08  =1 629  ;input, and the last buffer used by the GSC output.
02AF 08  =1 630  ;The control byte is defined as follows:
02B0 08  =1 631  ;|BIT 7|BIT 6|BIT 5|BIT 4|BIT 3|BIT 2|BIT 1|BIT 0|
02B1 08  =1 632  ;|-----------|
02B2 08  =1 633  |00 = BUFFER IA|
02B3 08  =1 634  |01 = BUFFER IB|
02B4 08  =1 635  |10 = BUFFER IC|
02B5 08  =1 636  |11 = BUFFER ID|
02B6 08  =1 637  |LAST BUFFER USED|
02B7 08  =1 638  |BY GSC FOR OUTPUT|
02B8 08  =1 639  |LAST BUFFER USED|
02B9 08  =1 640  |BY LSC FOR INPUT|
02BA 08  =1 641  |-----------|
02BB 08  =1 642  |
02BC 08  =1 643  |
02BD 08  =1 644  |
02BE 08  =1 645  |BIT 7 |BIT 6 |BIT 5 |BIT 4 |BIT 3 |BIT 2 |BIT 1 |BIT 0 |
02BF 08  =1 646  |
02C0 08  =1 647  |BUFFER IC |BUFFER IA|
02C1 08  =1 648  |ACTIVE |ACTIVE|
02C2 08  =1 649  |BUFFER ID |BUFFER IB|GSC_OUT_MSB|LSC_IN_MSB|
02C3 08  =1 650  |ACTIVE |ACTIVE|
02C4 08  =1 651  |BUFFER IC |BUF1A_ACT|GSC_OUT_LSB|LSC_IN_MSB|
02C5 08  =1 652  |
L0C OBJ  LINE  SOURCE
=1 653  BUFD_Act  BUFI_BACT
=1 654  
=1 655  
=1 656  JB LSC_IN_MSB, LSC_IN_ID_IA
=1 657  
=1 658  
=1 659  
=1 660  
=1 661  JB LSC_IN_LSB, LSC_IN_IC
=1 662  
=1 663  
=1 664  LSC_IN_IB
=1 665  
=1 666  
=1 667  
=1 668  JB BUFI_ACTIVE, BUFFERS_I_FULL
=1 669  
=1 670  
=1 671  
=1 672  MOV DPL, LOW (BUFI_ASTR_ADDR) - 3
=1 673  
=1 674  
=1 675  
=1 676  
=1 677  MOV A, IN_BYTE_COUNT
=1 678  
=1 679  
=1 680  
=1 681  INC DPTR
=1 682  
=1 683  
=1 684  MOV A, GSC_DEST_ADDR
=1 685  
=1 686  
=1 687  MOVX @DPTR, A
=1 688  
=1 689  INC DPTR
=1 690  
=1 691  
=1 692  MOV A, GSC_SRC_ADDR
=1 693  
=1 694  
=1 695  
=1 696  MOVX @DPTR, A
=1 697  
=1 698  
=1 699  
=1 700  
=1 701  CLR LSC_IN_MSB
=1 702  
=1 703  
=1 704  
=1 705  MOV LSC_INPUT_LOW, LOW (BUFI_BSTR_ADDR)
=1 706  
=1 707  
=1 708  MOV LSC_INPUT_HIGH, HIGH (BUFI_BSTR_ADDR)
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>028C</td>
<td>02032D</td>
<td>1</td>
<td>LSC_IN_IC:</td>
</tr>
<tr>
<td>028F</td>
<td>207E20</td>
<td>1</td>
<td>JB BUFIC_ACTIVE, BUFFERS_I_FULL: if buffer IC is active then the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>GSC has not yet emptied it and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>all the buffers must be full.</td>
</tr>
<tr>
<td>02C5</td>
<td>758300</td>
<td>1</td>
<td>MOV DPH, #HIGH (BUFIB_START_ADDR); setup DPTR to point at the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>beginning of buffer IB (first byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>should contain number of bytes</td>
</tr>
<tr>
<td>02C8</td>
<td>E57F</td>
<td>1</td>
<td>MOV A, IN_BYTE_COUNT; load acc with byte count for MOVX</td>
</tr>
<tr>
<td>02CA</td>
<td>F0</td>
<td>1</td>
<td>MOVX @DPTR, A; store byte count at first byte of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>buffer IB</td>
</tr>
<tr>
<td>02C8</td>
<td>A3</td>
<td>1</td>
<td>INC DPTR; D PTR now points to where the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>destination address should be</td>
</tr>
<tr>
<td>02CC</td>
<td>E57D</td>
<td>1</td>
<td>MOV A, GSC_DEST_ADDR; get stored destination address</td>
</tr>
<tr>
<td>02CE</td>
<td>F0</td>
<td>1</td>
<td>MOVX @DPTR, A; store destination addr in XRAM</td>
</tr>
<tr>
<td>02CF</td>
<td>A3</td>
<td>1</td>
<td>INC DPTR; D PTR now points to where source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>address should be stored</td>
</tr>
<tr>
<td>02D0</td>
<td>E57C</td>
<td>1</td>
<td>MOV A, GSC_SRC_ADDR; get stored source address</td>
</tr>
<tr>
<td>02D2</td>
<td>F0</td>
<td>1</td>
<td>MOVX @DPTR, A; store destination addr in XRAM</td>
</tr>
<tr>
<td>02D3</td>
<td>D27D</td>
<td>1</td>
<td>SETB BUFIB_ACTIVE; indicate that BUFIC has data to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>the output by the GSC and that the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>LSC has moved on to the next</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>buffer</td>
</tr>
<tr>
<td>02D7</td>
<td>C27B</td>
<td>1</td>
<td>CLR LSC_IN_LSB; reset flags to indicate that the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>current input buffer (for LSC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>is IC</td>
</tr>
<tr>
<td>02DF</td>
<td>02032D</td>
<td>1</td>
<td>JMP NEW_BUFI_IN_END; if the buffers are full, the pgm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>will be locked in the LSC service</td>
</tr>
</tbody>
</table>

APPNOT1

SOURCE

JMP NEW_BUFI_IN_END
LSC_IN_IC:
JB BUFIC_ACTIVE, BUFFERS_I_FULL: if buffer IC is active then the
GSC has not yet emptied it and
all the buffers must be full.
MOV DPH, #HIGH (BUFIB_START_ADDR); setup DPTR to point at the
beginning of buffer IB (first byte
should contain number of bytes.
MOV A, IN_BYTE_COUNT; load acc with byte count for MOVX
MOVX @DPTR, A; store byte count at first byte of
buffer IB
INC DPTR; D PTR now points to where the
destination address should be
MOV A, GSC_DEST_ADDR; get stored destination address
MOVX @DPTR, A; store destination addr in XRAM
INC DPTR; D PTR now points to where source
address should be stored
MOV A, GSC_SRC_ADDR; get stored source address
MOVX @DPTR, A; store destination addr in XRAM
SETB BUFIB_ACTIVE; indicate that BUFIC has data to
the output by the GSC and that the
LSC has moved on to the next
buffer
CLR LSC_IN_LSB; reset flags to indicate that the
current input buffer (for LSC) |
is IC
MOV LSC_INPUT_LOW, #LOW (BUFIC_START_ADDR)
MOV LSC_INPUT_HIGH, #HIGH (BUFIC_START_ADDR)
JMP NEW_BUFI_IN_END; if the buffers are full, the pgm
will be locked in the LSC service
LOC OBJ LINE SOURCE
=1 763
=1 764
=1 765
=1 766
=1 767
=1 768
=1 769
=1 770
=1 771
=1 772
=1 773
=1 774
=1 775
=1 776
=1 777
=1 778
=1 779
=1 780
=1 781
=1 782
=1 783
=1 784
=1 785
=1 786
=1 787
=1 788
=1 789
=1 790
=1 791
=1 792
=1 793
=1 794
=1 795
=1 796
=1 797
=1 798
=1 799
=1 800
=1 801
=1 802
=1 803
=1 804
=1 805
=1 806
=1 807
=1 808
=1 809
=1 810
=1 811
=1 812
=1 813
=1 814
=1 815
=1 816
=1 817

Routine in an "interrupt in progress" mode. If the DMA then frees up a buffer the interrupt routine cannot clear the buffer active bit until the interrupt (EOSTV/EOSTE) is serviced

Continue scanning active buffers until one is freed up

If LSC_IN = 1 then next buffer next buffer is 1A

If buffer ID is active then the OSC has not yet emptied it and all the buffers must be full

Setup DPTR to point at the beginning of buffer IC first byte should contain number of bytes

Load acc with byte count for MOVX

Store byte count at first byte of buffer IC

SDPTR now points to where the destination address should be

Get stored destination address

Store destination addr in XRAM

SDPTR now points to where source address should be stored

Get stored source address

Store destination addr in XRAM

Indicate that BUFIC has data to be output by the OSC and that the LSC has moved on to the next buffer

Set flags to indicate that the current input buffer (for LSC) is ID

MOV LSC_INPUT_LOW,#LOW (BUFID.START_ADDR)
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0307</td>
<td>757A01</td>
<td>=1</td>
<td>818 MOV LSC_INPUT_High, HIGH (BUF1A_STRT_ADDR)</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>819</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>820</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>821</td>
<td></td>
</tr>
<tr>
<td>030A</td>
<td>02032D</td>
<td>=1</td>
<td>822 JMP NEW_BUF1_IN_END</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>823</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>824</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>825</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>826</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>827</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>828</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>829</td>
<td></td>
</tr>
<tr>
<td>0310</td>
<td>758280</td>
<td>=1</td>
<td>830 MOV DPI, LOW (BUF1A_STRT_ADDR)</td>
</tr>
<tr>
<td>0313</td>
<td>758301</td>
<td>=1</td>
<td>831 MOV DPI, HIGH (BUF1A_STRT_ADDR)</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>832</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>833</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>834</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>835</td>
<td></td>
</tr>
<tr>
<td>0316</td>
<td>E57F</td>
<td>=1</td>
<td>836 MOV A, IN_BYTE_COUNT</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>837</td>
<td></td>
</tr>
<tr>
<td>0318</td>
<td>FO</td>
<td>=1</td>
<td>838 MOVX @DPTR.A</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>839</td>
<td></td>
</tr>
<tr>
<td>0319</td>
<td>A3</td>
<td>=1</td>
<td>840 INC DPTR</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>841</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>842</td>
<td></td>
</tr>
<tr>
<td>031A</td>
<td>E57D</td>
<td>=1</td>
<td>843 MOV A, GSC_DEST_ADDR</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>844</td>
<td></td>
</tr>
<tr>
<td>031C</td>
<td>FO</td>
<td>=1</td>
<td>845 MOVX @DPTR.A</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>846</td>
<td></td>
</tr>
<tr>
<td>031D</td>
<td>A3</td>
<td>=1</td>
<td>847 INC DPTR</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>848</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>849</td>
<td></td>
</tr>
<tr>
<td>031E</td>
<td>E57C</td>
<td>=1</td>
<td>850 MOV A, GSC_SRC_ADDR</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>851</td>
<td></td>
</tr>
<tr>
<td>0320</td>
<td>FO</td>
<td>=1</td>
<td>852 MOVX @DPTR.A</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>853</td>
<td></td>
</tr>
<tr>
<td>0321</td>
<td>D27F</td>
<td>=1</td>
<td>854 SETB BUF1D_ACTIVE</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>855</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>856</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>857</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>858</td>
<td></td>
</tr>
<tr>
<td>0323</td>
<td>C27B</td>
<td>=1</td>
<td>859 CLR LSC_IN_LSB</td>
</tr>
<tr>
<td>0325</td>
<td>C279</td>
<td>=1</td>
<td>860 CLR LSC_IN_MSB</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>861</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>862</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>863</td>
<td></td>
</tr>
<tr>
<td>0327</td>
<td>757B03</td>
<td>=1</td>
<td>864 MOV LSC_INPUT_LOW, LOW (BUF1A_STRT_ADDR)</td>
</tr>
<tr>
<td>032A</td>
<td>757A00</td>
<td>=1</td>
<td>865 MOV LSC_INPUT_High, HIGH (BUF1A_STRT_ADDR)</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>866</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>867</td>
<td></td>
</tr>
<tr>
<td>032D</td>
<td>22</td>
<td>=1</td>
<td>868 NEW_BUF1_IN_END:</td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>869</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>870</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>871</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=1</td>
<td>872</td>
<td></td>
</tr>
</tbody>
</table>
MCS-51 MACRO ASSEMBLER

LOC  OBJ  LINE  SOURCE

032E 32  =1 073  RETI  ; re-enable interrupts
         =1 074
         =1 075  NEW_BUFFER1_OUT
         =1 076
032F 30D903  =1 077  JNB  TEN.SECOND.TEN.CHECK  ; do not start another transmission
         =1 078  ; if one is in progress (signified
         =1 079  ; by TEN = 1) but this should never
         =1 080  ; happen
0332 0203AF  =1 081  TRANSMISSION_IN_PROGRESS  ; do not start a new GSC smt if one
         =1 082  ; is currently in progress
         =1 083
         =1 084
0335 20D9FA  =1 085  SECOND.TEN.CHECK  ; second one in case interrupt occurs during previous test
         =1 086
         =1 087  JB  TRANSMISSION_IN_PROGRESS
         =1 088
         =1 089  JB  GSC_OUT_MSB.GSC_OUT_IC_ID
         =1 090
         =1 091  JB  GSC_OUT_LSB.GSC_OUT_IB
         =1 092
         =1 093
         =1 094  GSC_OUT_IB:
         =1 095  =1 096
         =1 097  JNB  BUF1A_ACTIVE.NOTHING_FOR_GSC  ; if buffer 1A is not active then
         =1 098  ; the LSC has not yet filled it
         =1 099  ; since the GSC emptied it last
         =1 100
         =1 101  MOV DPTR,#(BUF1A_START_ADDR) -3  ; load DPTR with address of byte
         =1 102  ; that holds byte count for 1A
         =1 103
         =1 104  MOVX A,@DPTR  ; get byte count for buffer 1A
         =1 105
         =1 106  MOV BCRLO.A  ; load DMA byte count with length
         =1 107  ; of message to transmit
         =1 108
         =1 109  MOV BCRHO.0  ; ensure high byte count = 0
         =1 110  ; (should already be 0)
         =1 111
         =1 112  INC DPTR  ; DPTR now points at dest addr
         =1 113
         =1 114  MOV SARLO.DPL  ; source address for start of
         =1 115  ; data to send
         =1 116
         =1 117  MOV SARHO.DPH
         =1 118
0351 C278  =1 119  CLR GSC_OUT_MSB  ; indicate next output buffer will
         =1 120  ; be buffer 1B
0355 0203A6  =1 121  JMP START_GSC_OUT  ; routine that starts transmission
         =1 122
         =1 123  =1 124
         =1 125  GSC_OUT_IB:
         =1 126
         =1 127
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
<th>OPNDT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0358</td>
<td>307D54</td>
<td>1</td>
<td>JNB BUF1B_ACTIVE, NOTHING_FOR_GSC</td>
<td></td>
</tr>
<tr>
<td>0358</td>
<td>900080</td>
<td>1</td>
<td>MOV DPTR, # (BUF1B_STRCT_ADDR) -3</td>
<td></td>
</tr>
<tr>
<td>035E</td>
<td>E0</td>
<td>1</td>
<td>MOVX A, @DPTR</td>
<td></td>
</tr>
<tr>
<td>035F</td>
<td>F5E2</td>
<td>1</td>
<td>MOV BCRLO, A</td>
<td></td>
</tr>
<tr>
<td>0361</td>
<td>75E300</td>
<td>1</td>
<td>MOV BCRHO, #0</td>
<td></td>
</tr>
<tr>
<td>0364</td>
<td>A3</td>
<td>1</td>
<td>INC DPTR</td>
<td></td>
</tr>
<tr>
<td>0365</td>
<td>8582A2</td>
<td>1</td>
<td>MOV SARLO, DPL</td>
<td></td>
</tr>
<tr>
<td>0368</td>
<td>8583A3</td>
<td>1</td>
<td>MOV SARHO, DPH</td>
<td></td>
</tr>
<tr>
<td>036B</td>
<td>D27B</td>
<td>1</td>
<td>SETB GSC_OUT MSB</td>
<td></td>
</tr>
<tr>
<td>0370</td>
<td>C27A</td>
<td>1</td>
<td>CLR GSC_OUT_LSB</td>
<td></td>
</tr>
<tr>
<td>037F</td>
<td>0203A6</td>
<td>1</td>
<td>JMP START_GSC_OUT</td>
<td></td>
</tr>
<tr>
<td>0372</td>
<td>207A1A</td>
<td>1</td>
<td>GSC_OUT_IC_ID</td>
<td></td>
</tr>
<tr>
<td>0375</td>
<td>307E37</td>
<td>1</td>
<td>JNB BUF1C_ACTIVE, NOTHING_FOR_GSC</td>
<td></td>
</tr>
<tr>
<td>0378</td>
<td>900100</td>
<td>1</td>
<td>MOV DPTR, # (BUF1C_STRCT_ADDR) -3</td>
<td></td>
</tr>
<tr>
<td>037E</td>
<td>E0</td>
<td>1</td>
<td>MOVX A, @DPTR</td>
<td></td>
</tr>
<tr>
<td>037C</td>
<td>F5E2</td>
<td>1</td>
<td>MOV BCRLO, A</td>
<td></td>
</tr>
<tr>
<td>037E</td>
<td>75E300</td>
<td>1</td>
<td>MOV BCRHO, #0</td>
<td></td>
</tr>
<tr>
<td>0381</td>
<td>A3</td>
<td>1</td>
<td>INC DPTR</td>
<td></td>
</tr>
<tr>
<td>0382</td>
<td>8582A2</td>
<td>1</td>
<td>MOV SARLO, DPL</td>
<td></td>
</tr>
<tr>
<td>0385</td>
<td>8583A3</td>
<td>1</td>
<td>MOV SARHO, DPH</td>
<td></td>
</tr>
</tbody>
</table>

If buffer 1B is not active then:
- The LSC has not yet filled it since the GSC emptied it last.

If buffer 1C is not active then:
- The LSC has not yet filled it since the GSC emptied it last.

If GSC_OUT = 10B then the buffer is IC.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>038B</td>
<td>D27B</td>
<td>=1</td>
<td>983</td>
</tr>
<tr>
<td>038A</td>
<td>D27A</td>
<td>=1</td>
<td>984</td>
</tr>
<tr>
<td>038C</td>
<td>0203A6</td>
<td>=1</td>
<td>985</td>
</tr>
<tr>
<td>038F</td>
<td>307F1D</td>
<td>=1</td>
<td>998</td>
</tr>
<tr>
<td>0392</td>
<td>9001B0</td>
<td>=1</td>
<td>999</td>
</tr>
<tr>
<td>0395</td>
<td>E0</td>
<td>=1</td>
<td>1000</td>
</tr>
<tr>
<td>0396</td>
<td>F5E2</td>
<td>=1</td>
<td>1001</td>
</tr>
<tr>
<td>039B</td>
<td>A3</td>
<td>=1</td>
<td>1002</td>
</tr>
<tr>
<td>039C</td>
<td>8582A2</td>
<td>=1</td>
<td>1003</td>
</tr>
<tr>
<td>039D</td>
<td>8583A3</td>
<td>=1</td>
<td>1004</td>
</tr>
<tr>
<td>03A2</td>
<td>C27B</td>
<td>=1</td>
<td>1005</td>
</tr>
<tr>
<td>03A4</td>
<td>C27A</td>
<td>=1</td>
<td>1006</td>
</tr>
<tr>
<td>03A6</td>
<td>D2D9</td>
<td>=1</td>
<td>1007</td>
</tr>
<tr>
<td>03AB</td>
<td>D2C9</td>
<td>=1</td>
<td>1008</td>
</tr>
<tr>
<td>03A9</td>
<td>D2D0</td>
<td>=1</td>
<td>1009</td>
</tr>
<tr>
<td>03AC</td>
<td>439201</td>
<td>=1</td>
<td>1010</td>
</tr>
<tr>
<td>03AF</td>
<td>22</td>
<td>=1</td>
<td>1011</td>
</tr>
</tbody>
</table>

MCS-51 MACRO ASSEMBLER APPNOT1

LOC OBJ LINE SOURCE

038B D27B =1 983
038A D27A =1 984
038C 0203A6 =1 998
038F 307F1D =1 999
0392 9001B0 =1 1000
0395 E0 =1 1001
0396 F5E2 =1 1002
039B A3 =1 1003
039C 8582A2 =1 1004
039D 8583A3 =1 1005
03A2 C27B =1 1006
03A4 C27A =1 1007
03A6 D2D9 =1 1008
03AB D2C9 =1 1009
03A9 D2D0 =1 1010
03AC 439201 =1 1011
03AF 22 =1 1012

1: data to send
2: indicate next output buffer will be buffer ID
3: routine that starts transmission
4: if buffer ID is not active then the LSC has not yet filled it
5: since the GSC emptied it last
6: load DPTR with address of byte that holds byte count for ID
7: get byte count for buffer ID
8: load DMA byte count with length of message to transmit
9: source address for start of data to send
10: indicate next output buffer will be buffer 1A
11: routine that starts transmission
12: enable GSC transmitter
13: enable GSC transmit valid (TDN) interrupt
14: enable GSC transmit error int
15: start DMA which starts data output

270720-32
This section uses a bit addressable control byte to determine which buffers are active (contains data for LSC to output), the last buffer used by the LSC for output, and the last buffer used by the GSC for input.

The control byte is defined as follows:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **00** = BUFFER 2A
- **01** = BUFFER 2B
- **10** = BUFFER 2C
- **11** = BUFFER 2D

**LAST BUFFER USED BY GSC FOR INPUT**
**LAST BUFFER USED BY LSC FOR OUTPUT**

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**03B0 207343**
JB GSC_IN_MSB, GSC_IN_2D_2A

**03B1 20721E**
JB GSC_IN_LSB, GSC_IN_2C

**03B2 207639**
JB BUF2B_ACTIVE, BUFFERS_2_FULL

**03B3 750200**
MOV DPL, MLOW (BUF2A_START_ADDR) - 1

**03B4 750302**
MOV DPH, MHIGH (BUF2A_START_ADDR) - 1

**03B5 C3**
CLR C

**03C0 7476**
MOV A, #(MAX_LENGTH) - 2

**APPNOT1**
This section uses a bit addressable control byte to determine which buffers are active (contains data for LSC to output), the last buffer used by the LSC for output, and the last buffer used by the GSC for input. The control byte is defined as follows:

- **00** = BUFFER 2A
- **01** = BUFFER 2B
- **10** = BUFFER 2C
- **11** = BUFFER 2D

**LAST BUFFER USED BY GSC FOR INPUT**
**LAST BUFFER USED BY LSC FOR OUTPUT**

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**03B0 207343**
JB GSC_IN_MSB, GSC_IN_2D_2A  
if GSC_IN_MSB = 1 (2C or 2D), then the next buffer to be used must be 2D or 2A.

**03B1 20721E**
JB GSC_IN_LSB, GSC_IN_2C  
if GSC_IN = 01B then next buffer for GSC to use is 2C

**03B2 207639**
JB BUF2B_ACTIVE, BUFFERS_2_FULL  
if buffer 2B is active then the LSC has not yet emptied it and all the buffers must be full

**03B3 750200**
MOV DPL, MLOW (BUF2A_START_ADDR) - 1  
setup DPL to point at the beginning of buffer 2A (first byte should contain number of bytes)

**03B4 750302**
MOV DPH, MHIGH (BUF2A_START_ADDR) - 1  

**03B5 C3**
CLR C  
for SUBB

**03C0 7476**
MOV A, #(MAX_LENGTH) - 2  
maximum packet length and the initial value for BCRL (-2)
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>03C2</td>
<td>95F2</td>
<td>SUBB A, BCRL1</td>
</tr>
<tr>
<td></td>
<td>03C4</td>
<td>F0</td>
<td>MOVX @DPTR, A</td>
</tr>
<tr>
<td></td>
<td>03C5</td>
<td>D277</td>
<td>SETB BUF2A_ACTIVE</td>
</tr>
<tr>
<td></td>
<td>03C7</td>
<td>C273</td>
<td>CLR GSC_IN_MSB</td>
</tr>
<tr>
<td></td>
<td>03C9</td>
<td>D272</td>
<td>SETB GSC_IN_LSB</td>
</tr>
<tr>
<td></td>
<td>03C8</td>
<td>757901</td>
<td>MOV GSC_INPUT_LOW #LOW (BUF2B_START_ADDR)</td>
</tr>
<tr>
<td></td>
<td>03CE</td>
<td>757802</td>
<td>MOV GSC_INPUT_HIGH #HIGH (BUF2B_START_ADDR)</td>
</tr>
<tr>
<td></td>
<td>03D1</td>
<td>020432</td>
<td>JMP NEW_BUF2_IN_END</td>
</tr>
<tr>
<td></td>
<td>03D4</td>
<td>207518</td>
<td>JB BUF2C_ACTIVE, BUFFERS_2_FULL</td>
</tr>
<tr>
<td></td>
<td>03D7</td>
<td>758280</td>
<td>MOV DPL, #LOW (BUF2B_START_ADDR) - 1</td>
</tr>
<tr>
<td></td>
<td>03DA</td>
<td>758302</td>
<td>MOV DPH, #HIGH (BUF2B_START_ADDR)</td>
</tr>
<tr>
<td></td>
<td>03DD</td>
<td>C3</td>
<td>CLR C</td>
</tr>
<tr>
<td></td>
<td>03DE</td>
<td>7476</td>
<td>MOV A, #(MAX_LENGTH) - 2</td>
</tr>
<tr>
<td></td>
<td>03E0</td>
<td>95F2</td>
<td>SUBB A, BCRL1</td>
</tr>
<tr>
<td></td>
<td>03E2</td>
<td>F0</td>
<td>MOVX @DPTR, A</td>
</tr>
<tr>
<td></td>
<td>03E3</td>
<td>D276</td>
<td>SETB BUF2B_ACTIVE</td>
</tr>
</tbody>
</table>

- subtracted because first 2 bytes are the destination and source addresses
- load acc with byte count for MOVX
- store byte count at first byte of buffer 2A
- indicate that BUF2A has data to be output by the LSC and that the GSC has moved on to the next buffer
- reset flags to indicate that the current input buffer (for GSC) is 2B
- load starting address of buffer 2B
- if buffer 2C is active then the LSC has not yet emptied it and all the buffers must be full
- setup DPTR to point at the beginning of buffer 2B (first byte should contain number of bytes
- CLR C for SUBB
- maximum packet length and the initial value for BCRL1 (-2)
- subtracted because first 2 bytes are the destination and source addresses
- load acc with byte count for MOVX
- store byte count at first byte of buffer 2B
- indicate that BUF2B has data to be output by the LSC and that the GSC has moved on to the next buffer
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>03E5</td>
<td>0272</td>
<td>1148</td>
<td>CLR GSC_IN_LSB</td>
</tr>
<tr>
<td>03E7</td>
<td>0273</td>
<td>1149</td>
<td>SETB GSC_IN_MSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1150</td>
<td>iset flags to indicate that the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1151</td>
<td>current input buffer (for GSC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1152</td>
<td>is 2C</td>
</tr>
<tr>
<td>03E9</td>
<td>757901</td>
<td>1153</td>
<td>MOV GSC_INPUT_LOW,#LOW (BUF2C_START_ADDR)</td>
</tr>
<tr>
<td>03EC</td>
<td>757803</td>
<td>1154</td>
<td>MOV GSC_INPUT_HIGH,#HIGH (BUF2C_START_ADDR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1155</td>
<td>load starting address of buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1156</td>
<td>12C</td>
</tr>
<tr>
<td></td>
<td>020432</td>
<td>1157</td>
<td>JMP NEW_BUF2_IN_END</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1158</td>
<td>BUFFERS_2_FULL:</td>
</tr>
<tr>
<td>03F2</td>
<td>712E</td>
<td>1159</td>
<td>CALL IRET</td>
</tr>
<tr>
<td></td>
<td>1160</td>
<td>1161</td>
<td>if the buffers are full, the prog will be locked in the GSC service</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1162</td>
<td>routine in an &quot;interrupt in progress&quot; mode. If the DMA then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1163</td>
<td>frees up a buffer, the interrupt routine cannot clear the buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1164</td>
<td>active bit until the interrupt (ECS/RSV/ECSRD) is serviced</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1165</td>
<td></td>
</tr>
<tr>
<td></td>
<td>80BA</td>
<td>1166</td>
<td>JB GSC_INPUT_LOW#LOW (BUF2C_START_ADDR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1167</td>
<td>if GSC_IN = 1 then next buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1168</td>
<td>next buffer is 2A</td>
</tr>
<tr>
<td>03F6</td>
<td>20721E</td>
<td>1169</td>
<td>GSC_IN_2D:2A:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1170</td>
<td>JB GSC_IN_LSB,GSC_IN_2A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1171</td>
<td>if GSC_IN = 1 then next buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1172</td>
<td>next buffer is 2A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1173</td>
<td>GSC_IN_2D:2A:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1174</td>
<td>JB BUF2D_ACTIVE,BUFFERS_2_FULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1175</td>
<td>if buffer 2D is active then the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1176</td>
<td>GSC has not yet emptied it and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1177</td>
<td>all the buffers must be full</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1178</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2074F6</td>
<td>1179</td>
<td>MOV DPL,#LOW (BUF2C_START_ADDR) - 1</td>
</tr>
<tr>
<td></td>
<td>758200</td>
<td>1180</td>
<td>MOV DPL,#HIGH (BUF2C_START_ADDR)</td>
</tr>
<tr>
<td></td>
<td>758303</td>
<td>1181</td>
<td>$ ;setup DPTR to point at the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1182</td>
<td>$ ;beginning of buffer 2C (first byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1183</td>
<td>$ ;should contain number of bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1184</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1185</td>
<td>1186</td>
<td>MOVX @DPTR,A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1187</td>
<td>store byte count at first byte of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1188</td>
<td>buffer 2C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1189</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1190</td>
<td>1191</td>
<td>MOV A,#(MAX_LENGTH) - 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1192</td>
<td>maximum packet length and the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1193</td>
<td>initial value for BCR1 ( 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1194</td>
<td>subtracted because first 2 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1195</td>
<td>are the destination and source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1196</td>
<td>addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1197</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1198</td>
<td>1199</td>
<td>SUBB A,BCR1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1200</td>
<td>load acc with byte count for MOVX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1201</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1202</td>
<td>1203</td>
<td>SETB BUF2C_ACTIVE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1204</td>
<td>indicate that BUF2C has data to</td>
</tr>
<tr>
<td>LOC OBJ</td>
<td>LINE</td>
<td>SOURCE</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>---------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>=1 1203</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1204</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1205</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1206</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>040A D272</td>
<td>=1</td>
<td>SETB GSC_IN_LSB</td>
<td></td>
</tr>
<tr>
<td>040C D273</td>
<td>=1</td>
<td>SETB GSC_IN_MSB</td>
<td></td>
</tr>
<tr>
<td>=1 1207</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1208</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1209</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1210</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>040E 757981</td>
<td>=1</td>
<td>MOV GSC_INPUT_LOW, LOW (BUF2D_STRT_ADDR)</td>
<td></td>
</tr>
<tr>
<td>0411 757803</td>
<td>=1</td>
<td>MOV GSC_INPUT_HIGH, HIGH (BUF2D_STRT_ADDR)</td>
<td></td>
</tr>
<tr>
<td>=1 1211</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1212</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1213</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1214</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0414 020432</td>
<td>=1</td>
<td>JMP NEW_BUF2_IN_END</td>
<td></td>
</tr>
<tr>
<td>=1 1216</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1217</td>
<td></td>
<td>GSC_IN_2A</td>
<td></td>
</tr>
<tr>
<td>=1 1218</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1219</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0417 207708</td>
<td>=1</td>
<td>JB BUF2A_ACTIVE, BUFFERS_2_FULL</td>
<td></td>
</tr>
<tr>
<td>=1 1220</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1221</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1222</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1223</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>041A 758280</td>
<td>=1</td>
<td>MOV DPH, LOW (BUF2D_STRT_ADDR) - 1</td>
<td></td>
</tr>
<tr>
<td>041D 758303</td>
<td>=1</td>
<td>MOV DPH, HIGH (BUF2D_STRT_ADDR)</td>
<td></td>
</tr>
<tr>
<td>=1 1224</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1225</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1226</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1227</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1228</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0420 C3</td>
<td>=1</td>
<td>CLR C</td>
<td></td>
</tr>
<tr>
<td>=1 1229</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1230</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0421 7476</td>
<td>=1</td>
<td>MOV A, #(MAX_LENGTH) - 2</td>
<td></td>
</tr>
<tr>
<td>=1 1231</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1232</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1233</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1234</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1235</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1236</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1237</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1238</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0425 F0</td>
<td>=1</td>
<td>MOVX @DPTR, A</td>
<td></td>
</tr>
<tr>
<td>=1 1239</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1240</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1241</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0426 D274</td>
<td>=1</td>
<td>SETB BUF2D_ACTIVE</td>
<td></td>
</tr>
<tr>
<td>=1 1242</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1243</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1244</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1245</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1246</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0428 C272</td>
<td>=1</td>
<td>CLR GSC_IN_LSB</td>
<td></td>
</tr>
<tr>
<td>042A C273</td>
<td>=1</td>
<td>CLR GSC_IN_MSB</td>
<td></td>
</tr>
<tr>
<td>=1 1247</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1248</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1249</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1250</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>042C 757901</td>
<td>=1</td>
<td>MOV GSC_INPUT_LOW, LOW (BUF2A_STRT_ADDR)</td>
<td></td>
</tr>
<tr>
<td>042F 757802</td>
<td>=1</td>
<td>MOV GSC_INPUT_HIGH, HIGH (BUF2A_STRT_ADDR)</td>
<td></td>
</tr>
<tr>
<td>=1 1251</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1252</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1253</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1254</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1255</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=1 1257</td>
<td></td>
<td>NEW_BUF2_IN_END</td>
<td></td>
</tr>
</tbody>
</table>
MOV DARL1,GSC_INPUT_LOW
MOV DARH1,GSC_INPUT_HIGH

load DMA destination address
registers with starting address
of current buffer area

MOV BCRH,#0
MOV BCRH,#MAX_LENGTH

load DMA byte count with packet length

RET

NEW_BUFFER2_OUT:

LSC_XMIT_IN_PROGRESS:

JMP NOTHING_FOR_LSC

do not start another transmission
if one is in progress (signified
by LSC_ACTIVE = 1) but this
should never happen

LSC_XMIT_IN_PROGRESS:

SECOND_LSC_CHECK:

JB LSC_ACTIVE,SECOND_LSC_CHECK
second one in case interrupt
occurs during previous test

JB LSC_OUT_MSB,LSC_OUT_RC_2D
if LSC_OUT_MSB = 1 then current
buffer is 2C or 2D

JB LSC_OUT_LSB,LSC_OUT_RC_2B
if LSC_OUT_LSB = 1 then current
buffer is 2B

LSC_XMIT_OUT_2A:

JNB BUF2A_ACTIVE,NOTHING_FOR_LSC
if buffer 2A is not active then
the GSC has not yet filled it
since the LSC emptied it last

SETB LSC_ACTIVE

show that LSC is in the process of

MOV DPTR,#(BUF2A_STRT_ADDR) -1
load DPTR with address of byte
that holds byte count for 2A

MOVI A,#DPTR
get byte count for buffer 2A

MOV LSC_OUT_COUNTER,A
load LSC byte counter with length
of message to transmit

INC LSC_OUT_COUNTER
incremented because the counter
is first decremented before being
tested (DJNZ) when LSC begins to
output data
LINE
=1131
=1132
=1133
=1134
=1135
=1136
=1137
=1138
=1139
=1140
=1141
=1142
=1143
=1144
=1145
=1146
=1147
=1148
=1149
=1150
=1151
=1152
=1153
=1154
=1155
=1156
=1157
=1158
=1159
=1160
=1161
=1162
=1163
=1164
=1165
=1166
=1167
=1168
=1169
=1170
=1171
=1172
=1173
=1174
=1175
=1176
=1177
=1178
=1179
=1180
=1181
=1182
=1183
=1184
=1185
=1186
=1187
=1188
=1189
=1190
=1191
=1192
=1193
=1194
=1195
=1196
=1197
=1198
=1199
=1200
=1201
=1202
=1203
=1204
=1205
=1206
=1207
=1208
=1209
=1210
=1211
=1212
=1213
=1214
=1215
=1216
=1217
=1218
=1219
=1220
=1221
=1222
=1223
=1224
=1225
=1226
=1227
=1228
=1229
=1230
=1231
=1232
=1233
=1234
=1235
=1236
=1237
=1238
=1239
=1240
=1241
=1242
=1243
=1244
=1245
=1246
=1247
=1248
=1249
=1250
=1251
=1252
=1253
=1254
=1255
=1256
=1257
=1258
=1259
=1260
=1261
=1262
=1263
=1264
=1265
=1266
=1267

indicate next output buffer will be buffer 2B
routine that starts transmission
if LSC_OUT = 01B then the buffer is 2B
if buffer 2B is not active then the SGC has not yet filled it since the LSC emptied it last
show that LSC is in the process of doing a transmission
load DPTR with address of byte that holds byte count for 2B
get byte count for buffer 2B
load LSC byte counter with length of message to transmit incremented because the counter is first decremented before being tested (DJNZ) when LSC begins to output data
indicate next output buffer will be buffer 2C
routine that starts transmission
if LSC_OUT = 11B then current buffer is 2D
if LSC_OUT = 10B then the buffer is 2C
if buffer 2C is not active then the SGC has not yet filled it since the LSC emptied it last
show that LSC is in the process of doing a transmission
load DPTR with address of byte that holds byte count for 2C
get byte count for buffer 2C~
MOV LSC_OUT_COUNTER.A
INC LSC_OUT_COUNTER
SETB LSC_OUT_MSB
SETB LSC_OUT_LSB
JMP START_LSC_OUT
LSC_OUT_2D:
JNB BUF2D_ACTIVE, NOTHING_FOR_LSC
SETB LSC_ACTIVE
MOV DPTR,#(BUF2D_STRT_ADDR)-1
MOVX A, @DPTR
INC LSC_OUT_COUNTER.A
CLRA
INC DPTR
INC DPTR
INC DPTR
MOV LSC_OUTPUT_LOW.DPL
MOV LSC_OUTPUT_HIGH.DPH
START_LSC_OUT:
INC DPTR
INC DPTR
INC DPTR
INC DPTR
MOV LSC_OUTPUT_LOW.DPL
MOV LSC_OUTPUT_HIGH.DPH
APPND1

SOURCE

**MCS-51 MACRO ASSEMBLER**

LINE  

LOC OBJ

=1 1423
=1 1424
=1 1425
=1 1426
=1 1427
=1 1428
=1 1429
=1 1430
=1 1431
=1 1432
=1 1433
=1 1434
=1 1435
=1 1436
=1 1437
=1 1438
=1 1439
=1 1440
=1 1441
=1 1442
=1 1443
=1 1444
=1 1445
=1 1446
=1 1447
=1 1448
=1 1449
=1 1450
=1 1451
=1 1452
=1 1453
=1 1454
=1 1455
=1 1456
=1 1457
=1 1458
=1 1459
=1 1460
=1 1461
=1 1462
=1 1463
=1 1464
=1 1465
=1 1466
=1 1467
=1 1468
=1 1469
=1 1470
=1 1471
=1 1472
=1 1473
=1 1474
=1 1475
=1 1476
=1 1477

; to send
 SETB TI ; set interrupt flag to start

; transmitting when main program is returned to

NOTHING FOR LSC:

RET

+1 *INCLUDE (XMITVAL.SRC)

GSC_VALID_XMIT:

PUSH DPL
PUSH DPH
PUSH ACC
PUSH PSW

; SFRs to save before servicing interrupt

; ***********************

CLR EI:STV
CLR EXT

CLEAR_ACTIVE_BUFFER:

JB GSC_OUT_MSB,CLEAR_ACTIVE_ID

; if GSC_OUT_MSB = 1 then

; previous used buffer for GSC output must have been ID or IC

CLEAR_ACTIVE_ID:

JB GSC_OUT_LSB,CLEAR_ACTIVE_ID

; if GSC_OUT = 01 then active buffer ID bit must be cleared

CLEAR_ACTIVE_IA:

JB FIRST_GSC_OUT,END_CLEARACTIVE

; if this is first transmission, do not clear buffer ID active bit (this may happen if all four buffers are filled before first GSC transmission)

CLR BUFID_ACTIVE

; if GSC_OUT = 00, then last buffer used is ID unless first transmission

JMP END_CLEAR_ACTIVE_OUT

CLEAR_ACTIVE_IA:
<table>
<thead>
<tr>
<th>LOC OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>04C4 C27C</td>
<td>1478</td>
<td>CLR BUF1A_ACTIVE</td>
</tr>
<tr>
<td>04C4 C27C</td>
<td>1479</td>
<td>; if GSC_OUT = 01, then last buffer used is 1A</td>
</tr>
<tr>
<td>04C4 C27C</td>
<td>1480</td>
<td>; buffer used is 1A</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1481</td>
<td>CLR FIRST_GSC_OUT</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1482</td>
<td>; clear indicator that shows the first GSC transmission has not yet occurred</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1483</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1484</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1485</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1486</td>
<td>JMP END_CLEAR_ACTIVE_OUT</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1487</td>
<td>CLEAR_ACTIVE_IC:</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1488</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1489</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1490</td>
<td>JB GSC_OUT_LSB,CLEAR_ACTIVE_IC</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1491</td>
<td>; if GSC_OUT = 11B, then last buffer used is 1C</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1492</td>
<td>; buffer used is 1C</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1493</td>
<td>CLEAR_ACTIVE_IC:</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1494</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1495</td>
<td>CLR BUF1B_ACTIVE</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1496</td>
<td>; if GSC_OUT = 10, then last buffer used is 1B</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1497</td>
<td>; buffer used is 1B</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1498</td>
<td>JMP END_CLEAR_ACTIVE_OUT</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1499</td>
<td>CLEAR_ACTIVE_IC:</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1501</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1502</td>
<td>CLR BUF1C_ACTIVE</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1503</td>
<td>; if GSC_OUT = 11, then last buffer used is 1C unless first transmission</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1504</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1505</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1506</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1507</td>
<td>END_CLEAR_ACTIVE_OUT:</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1508</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1509</td>
<td>;*******************************************************************************</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1510</td>
<td>; see if next buffer is full or INIT ADDRESS FOR NEXT AVAIL BUFFER</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1511</td>
<td>; WHEN IT IS FILLED</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1512</td>
<td>;*******************************************************************************</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1513</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1514</td>
<td>CALL NEW_BUFFER1_OUT</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1515</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1516</td>
<td>;*******************************************************************************</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1517</td>
<td>; RETURN TO MAIN PROGRAM LOOP</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1518</td>
<td>;*******************************************************************************</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1519</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1520</td>
<td>MOV TDCNT.#0</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1521</td>
<td>; clear collision counter</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1522</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1523</td>
<td>POP PSW</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1524</td>
<td>POP ACC</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1525</td>
<td>POP DPH</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1526</td>
<td>POP DPL</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1527</td>
<td>RETI</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1528</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1529+</td>
<td>INCLUDE (XMITERR.SRC)</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1530</td>
<td>GSC_ERROR_XMIT:</td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1531</td>
<td></td>
</tr>
<tr>
<td>04C6 C26F</td>
<td>1532</td>
<td>;*******************************************************************************</td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ</td>
<td>LINE</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>04E3</td>
<td>5392FE</td>
<td>1</td>
</tr>
<tr>
<td>04E3</td>
<td>5392FE</td>
<td>1</td>
</tr>
<tr>
<td>04E3</td>
<td>5392FE</td>
<td>1</td>
</tr>
<tr>
<td>04E3</td>
<td>5392FE</td>
<td>1</td>
</tr>
<tr>
<td>04E6</td>
<td>C082</td>
<td>1</td>
</tr>
<tr>
<td>04E6</td>
<td>C083</td>
<td>1</td>
</tr>
<tr>
<td>04EA</td>
<td>C0E0</td>
<td>1</td>
</tr>
<tr>
<td>04EC</td>
<td>C0E0</td>
<td>1</td>
</tr>
<tr>
<td>04EC</td>
<td>C0E0</td>
<td>1</td>
</tr>
<tr>
<td>04EC</td>
<td>C0E0</td>
<td>1</td>
</tr>
<tr>
<td>04EC</td>
<td>C0E0</td>
<td>1</td>
</tr>
<tr>
<td>04EC</td>
<td>C0E0</td>
<td>1</td>
</tr>
<tr>
<td>04F1</td>
<td>7BFF</td>
<td>1</td>
</tr>
<tr>
<td>04F1</td>
<td>7BFF</td>
<td>1</td>
</tr>
<tr>
<td>04F3</td>
<td>020500</td>
<td>1</td>
</tr>
<tr>
<td>04F3</td>
<td>020500</td>
<td>1</td>
</tr>
<tr>
<td>04F3</td>
<td>020500</td>
<td>1</td>
</tr>
<tr>
<td>04F3</td>
<td>020500</td>
<td>1</td>
</tr>
<tr>
<td>04F3</td>
<td>020500</td>
<td>1</td>
</tr>
<tr>
<td>04F6</td>
<td>30E05</td>
<td>1</td>
</tr>
<tr>
<td>04F6</td>
<td>30E05</td>
<td>1</td>
</tr>
<tr>
<td>04F6</td>
<td>30E05</td>
<td>1</td>
</tr>
<tr>
<td>04F9</td>
<td>7BDFS</td>
<td>1</td>
</tr>
<tr>
<td>04F9</td>
<td>7BDFS</td>
<td>1</td>
</tr>
<tr>
<td>04F9</td>
<td>7BDFS</td>
<td>1</td>
</tr>
<tr>
<td>04FB</td>
<td>020500</td>
<td>1</td>
</tr>
<tr>
<td>04FB</td>
<td>020500</td>
<td>1</td>
</tr>
<tr>
<td>04FE</td>
<td>7BDFS</td>
<td>1</td>
</tr>
<tr>
<td>04FE</td>
<td>7BDFS</td>
<td>1</td>
</tr>
<tr>
<td>0500</td>
<td>5175</td>
<td>1</td>
</tr>
<tr>
<td>0500</td>
<td>5175</td>
<td>1</td>
</tr>
<tr>
<td>0500</td>
<td>5175</td>
<td>1</td>
</tr>
<tr>
<td>0502</td>
<td>E52F</td>
<td>1</td>
</tr>
<tr>
<td>0504</td>
<td>540E</td>
<td>1</td>
</tr>
<tr>
<td>0504</td>
<td>540E</td>
<td>1</td>
</tr>
<tr>
<td>0504</td>
<td>540E</td>
<td>1</td>
</tr>
<tr>
<td>0506</td>
<td>840012</td>
<td>1</td>
</tr>
<tr>
<td>0506</td>
<td>840012</td>
<td>1</td>
</tr>
<tr>
<td>0506</td>
<td>840012</td>
<td>1</td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ</td>
<td>LINE</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>0509</td>
<td>75A203</td>
<td>=1</td>
</tr>
<tr>
<td>050C</td>
<td>75A300</td>
<td>=1</td>
</tr>
<tr>
<td>050F</td>
<td>900000</td>
<td>=1</td>
</tr>
<tr>
<td>0512</td>
<td>E0</td>
<td>=1</td>
</tr>
<tr>
<td>0513</td>
<td>F5E2</td>
<td>=1</td>
</tr>
<tr>
<td>0515</td>
<td>75E300</td>
<td>=1</td>
</tr>
<tr>
<td>0518</td>
<td>020554</td>
<td>=1</td>
</tr>
<tr>
<td>051B</td>
<td>B40912</td>
<td>=1</td>
</tr>
<tr>
<td>051E</td>
<td>75A203</td>
<td>=1</td>
</tr>
<tr>
<td>0520</td>
<td>900000</td>
<td>=1</td>
</tr>
<tr>
<td>0522</td>
<td>E0</td>
<td>=1</td>
</tr>
<tr>
<td>0525</td>
<td>F5E2</td>
<td>=1</td>
</tr>
<tr>
<td>052A</td>
<td>75E300</td>
<td>=1</td>
</tr>
<tr>
<td>052D</td>
<td>020554</td>
<td>=1</td>
</tr>
<tr>
<td>0530</td>
<td>B40912</td>
<td>=1</td>
</tr>
<tr>
<td>0533</td>
<td>75A203</td>
<td>=1</td>
</tr>
<tr>
<td>0536</td>
<td>75A301</td>
<td>=1</td>
</tr>
<tr>
<td>0539</td>
<td>900100</td>
<td>=1</td>
</tr>
</tbody>
</table>

**SOURCE**

- Pre-initialize source pointer into BUF1A
- Allocation that holds BUF1A byte count
- Pre-initialize byte counter with number of bytes in BUF1A
- Pre-initialize source pointer into BUFIB
- Allocation that holds BUFIB byte count
- Pre-initialize byte counter with number of bytes in BUF1B
- Pre-initialize source pointer into BUFIC
- Allocation that holds BUFIC byte count
- Pre-initialize byte counter with number of bytes in BUF1C

---

**APPNOTES**

- MOV SARLO. #LOW (BUFIA_STRT_ADDR)
- MOV SARHO. #HIGH (BUFIA_STRT_ADDR)
- MOV DPTR. # (BUFIA_STRT_ADDR) -3
- MOVX A, @DPTR
- MOV BCRLO. A
- MOV BCRHO. #0
- MOV SARLO. #LOW (BUFIB_STRT_ADDR)
- MOV SARHO. #HIGH (BUFIB_STRT_ADDR)
- MOV DPTR. # (BUFIB_STRT_ADDR) -3
- MOVX A, @DPTR
- MOV BCRLO. A
- MOV BCRHO. #0
- MOV SARLO. #LOW (BUFFC_STRT_ADDR)
- MOV SARHO. #HIGH (BUFFC_STRT_ADDR)
- MOV DPTR. # (BUFFC_STRT_ADDR) -3
- MOVX A, @DPTR
- MOV BCRLO. A
- MOV BCRHO. #0
MCS-51 MACRO ASSEMBLER APPNOT

LOC OBJ LINE SOURCE
0542 0202554 =1 1643 JMP START_RETRANSMIT
0542 0202554 =1 1644
0542 0202554 =1 1645
0542 0202554 =1 1646 BUFFERID_RELOAD
0542 0202554 =1 1647
0545 75A2B3 =1 1648 MOV SARLO. LOW (BUF1D_STRT_ADDR)
0546 75A2B3 =1 1649 MOV SARHO. HIGH (BUF1D_STRT_ADDR)
0546 75A2B3 =1 1650 .re-initialize source pointer
0546 75A2B3 =1 1651 .to BUF1D
0546 75A2B3 =1 1652 .location that holds BUF1D
0546 75A2B3 =1 1653 MOV DPT1. BUF1D_STRT_ADDR -3
0546 75A2B3 =1 1654 .byte count
0546 75A2B3 =1 1655
0546 75A2B3 =1 1656 MOVX A. @DPTR
0546 75A2B3 =1 1657 .get byte count
0546 75A2B3 =1 1658 MOV BCRLO. A
0546 75A2B3 =1 1659 .re-initialize byte counter
0546 75A2B3 =1 1660 .with number of bytes in BUF1A
0546 75A2B3 =1 1661
0546 75A2B3 =1 1662
0546 75A2B3 =1 1663
0546 75A2B3 =1 1664 :================================================================================
0546 75A2B3 =1 1665 . ENABLE TRANSMITTER AND DMA CHANNEL
0546 75A2B3 =1 1666 :==========================================================================
0546 75A2B3 =1 1667 MOV TCDCTN. #0
0546 75A2B3 =1 1668 .clear collision counter
0546 75A2B3 =1 1669
0547 02D9 =1 1670 SETB TEN
0547 02D9 =1 1671
0549 30B9FD =1 1672 JNB TEN. $;
0549 30B9FD =1 1673 .wait until TEN is set (TEN
0549 30B9FD =1 1674 .will not be set if a
0549 30B9FD =1 1675 .transmissions CRC has not yet
0549 30B9FD =1 1676 .completed but TEN might be
0549 30B9FD =1 1677 .cleared before CRC completes)
054C 439201 =1 1678 ORL DCONO. #01
054C 439201 =1 1679 .set GO bit
054F 0080 =1 1680 POP PSW
0561 00E0 =1 1681 POP ACC
0563 0033 =1 1682 POP DPH
0565 0082 =1 1683 POP DPL
0567 32 =1 1684 RETI
0567 32 =1 1685
0567 32 =1 1686 +1 INCLUDE (RECVAL.SRC)
0567 32 =1 1686 .SFRs that were saved
0567 32 =1 1686
0567 32 =1 1687 .SFRs to save before servicing
0567 32 =1 1687 .interrupt
0570 7180 =1 1696 CALL NEW_BUFFER2_IN
0570 7180 =1 1697 .save byte count, select next
0570 7180 =1 1697 .GSC input buffer, setup next
0570 7180 =1 1697
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0570</td>
<td>D0E0</td>
<td>=1</td>
<td>0082 ! destination address, and</td>
</tr>
<tr>
<td>0570</td>
<td>D0E1</td>
<td>=1</td>
<td>0083 ! setup new byte count</td>
</tr>
<tr>
<td>0571</td>
<td>D0E2</td>
<td>=1</td>
<td>008A ! set GO bit for DMA 1</td>
</tr>
<tr>
<td>0572</td>
<td>D0E3</td>
<td>=1</td>
<td>008B ! enable receiver</td>
</tr>
<tr>
<td>0573</td>
<td>D0E4</td>
<td>=1</td>
<td>008C ! SFRs that were saved</td>
</tr>
<tr>
<td>0574</td>
<td>D0E5</td>
<td>=1</td>
<td>008D ! SFRs to save before servicing</td>
</tr>
<tr>
<td>0575</td>
<td>D0E6</td>
<td>=1</td>
<td>008E ! interrupt</td>
</tr>
<tr>
<td>0576</td>
<td>D0E7</td>
<td>=1</td>
<td>008F !</td>
</tr>
<tr>
<td>0577</td>
<td>D0E8</td>
<td>=1</td>
<td>0090 !</td>
</tr>
<tr>
<td>0578</td>
<td>D0E9</td>
<td>=1</td>
<td>0091 !</td>
</tr>
<tr>
<td>0579</td>
<td>D0EA</td>
<td>=1</td>
<td>0092 !</td>
</tr>
<tr>
<td>0580</td>
<td>D0EB</td>
<td>=1</td>
<td>0093 !</td>
</tr>
<tr>
<td>0581</td>
<td>D0EC</td>
<td>=1</td>
<td>0094 !</td>
</tr>
<tr>
<td>0582</td>
<td>D0ED</td>
<td>=1</td>
<td>0095 !</td>
</tr>
<tr>
<td>0583</td>
<td>D0EE</td>
<td>=1</td>
<td>0096 !</td>
</tr>
<tr>
<td>0584</td>
<td>D0EF</td>
<td>=1</td>
<td>0097 !</td>
</tr>
<tr>
<td>0585</td>
<td>D0F0</td>
<td>=1</td>
<td>0098 !</td>
</tr>
<tr>
<td>0586</td>
<td>D0F1</td>
<td>=1</td>
<td>0099 !</td>
</tr>
<tr>
<td>0587</td>
<td>D0F2</td>
<td>=1</td>
<td>009A !</td>
</tr>
<tr>
<td>0588</td>
<td>D0F3</td>
<td>=1</td>
<td>009B !</td>
</tr>
<tr>
<td>0589</td>
<td>D0F4</td>
<td>=1</td>
<td>009C !</td>
</tr>
<tr>
<td>058A</td>
<td>D0F5</td>
<td>=1</td>
<td>009D !</td>
</tr>
<tr>
<td>058B</td>
<td>D0F6</td>
<td>=1</td>
<td>009E !</td>
</tr>
<tr>
<td>058C</td>
<td>D0F7</td>
<td>=1</td>
<td>009F !</td>
</tr>
<tr>
<td>058D</td>
<td>D0F8</td>
<td>=1</td>
<td>00A0 !</td>
</tr>
<tr>
<td>058E</td>
<td>D0F9</td>
<td>=1</td>
<td>00A1 !</td>
</tr>
<tr>
<td>058F</td>
<td>D0FA</td>
<td>=1</td>
<td>00A2 !</td>
</tr>
<tr>
<td>0590</td>
<td>D0FB</td>
<td>=1</td>
<td>00A3 !</td>
</tr>
<tr>
<td>0591</td>
<td>D0FC</td>
<td>=1</td>
<td>00A4 !</td>
</tr>
<tr>
<td>0592</td>
<td>D0FD</td>
<td>=1</td>
<td>00A5 !</td>
</tr>
<tr>
<td>0593</td>
<td>D0FE</td>
<td>=1</td>
<td>00A6 !</td>
</tr>
<tr>
<td>0594</td>
<td>D0FF</td>
<td>=1</td>
<td>00A7 !</td>
</tr>
</tbody>
</table>

**SOURCE**

```
ORL DCON, #01
SETB GREN
POP PSW
POP ACC
POP DPH
POP DPL

#define _INCLUDE SRC
GSC, ERROR_REC
PUSH DPL
PUSH DPH
PUSH ACC
PUSH PSW

*******************************************************************************
** index error type
*******************************************************************************

INC_ERROR_COUNT:
*******************************************************************************
** increment the error counter
*******************************************************************************

RCABT CHECK:
*******************************************************************************
** see if error caused by RCABT
*******************************************************************************

JNB RCABT, OVR_CHECK
MOV ERROR POINTER, RCABT_COUNTER
CALL INCREMENT_COUNTER
JMP REC_ERROR_COUNT_END
```
MCS-51 MACRO ASSEMBLER

LOC OBJ LINE SOURCE

0592 30EF07 = 1 1753 DVR_CHECK: 
  = 1 1754 JNB DVR.CRC_CHECK 
  = 1 1755 see if error caused by DVR

0595 78F9 = 1 1756 MOV ERROR_POINTER.#DVR_COUNTER

0597 5175 = 1 1757 CALL INCREMENT_COUNTER

0599 0205AA = 1 1758 JMP REC_ERROR_COUNT_END

059C 30EC07 = 1 1759 CRC_CHECK 
  = 1 1760 see if error caused by CRCE

059F 7BE7 = 1 1761 MOV ERROR_POINTER.#CRCE_COUNTER

05A1 5175 = 1 1762 CALL INCREMENT_COUNTER

05A3 0205AA = 1 1763 JMP REC_ERROR_COUNT_END

05A6 7BED = 1 1764 ERR._POINTER, #AE_COUNTER 
  = 1 1765 only error type left

05A8 5175 = 1 1766 CALL INCREMENT_COUNTER

05A9 REC_ERROR_COUNT_END 
  = 1 1767

05AA 71B0 = 1 1768 .this is not what I want to do probably. I may need to fool with current 
  = 1 1769 active bit addressing byte count or who knows what???? 
  = 1 1770

05AC 439301 = 1 1771 CALL NEW_BUFFER_IN 
  = 1 1772 say what this routine does

05AF D2E9 = 1 1773 ORL DCON1,#01 
  = 1 1774 set GO bit for DMA1

05B1 D0D0 = 1 1775 SETB OREN 
  = 1 1776 enable receiver

05B3 0DB3 = 1 1777 POP PSW

05B5 DB82 = 1 1778 POP DPH

05B9 32 = 1 1779 RETI

05C2 30980C = 1 1780 JNB RI.XMIT_LSC 
  = 1 1781 jump to LSC transmit service

05C5 1205DD = 1 1782 CALL LSC_RECEIVE 
  = 1 1783 invoke LSC receiver server

270720-46
APPNOTE

SOURCE

RETIR

XMIT_LSC

0501 1205FF = 1 1817 CALL LSC_XMIT...invoke LSC transmit server

0504 D000 = 1 1818 POP PSW...SFRs that were saved

0506 D000 = 1 1820 POP ACC...SFRs that were saved

0509 D000 = 1 1822 POP DPH...return from interrupt

0512 32 = 1 1823 POP DPL...return from interrupt

0515 32 = 1 1824 RETI...return from interrupt

051E 05FF = 1 1825 LSC_RECEIVE:

0520 C29B = 1 1826 CLR RI...clear receiver interrupt bit

0524 05FF = 1 1827 INC IN_BYTE_COUNT...increment RAM location that accounts the number of bytes input from LSC

0528 05FF = 1 1831 MOV DPL, LSC_INPUT_LOW...target address where next byte received by LSC will be stored

052C 05FF = 1 1834 MOV DPH, LSC_INPUT_HIGH...target oldest byte LSC has received

0530 E599 = 1 1837 MOV A, SBUF...store byte in buffer

0534 F0 = 1 1838 MOVX @DPTR, A...store incremented buffer address

0538 A3 = 1 1840 INC DPH...increment buffer address

053C 85278 = 1 1841 MOV LSC_INPUT_LOW, DPL...store incremented address

053E 85279 = 1 1843 MOV LSC_INPUT_HIGH, DPH...initialize for next buffer

0542 8400A = 1 1844 CJNE A, #CR.END_LSC_RECEIVE...if last character received was an ASCII carriage return

0546 05FF = 1 1847 INC IN_BYTE_COUNT...increment RAM location that accounts the number of bytes input from LSC

054A 740A = 1 1850 MOV A, #LINE_FEED...insert a line feed after the carriage return for QSC to transmit

054E 05FF = 1 1852 INC IN_BYTE_COUNT...increment RAM location that accounts the number of bytes input from LSC

0552 740A = 1 1853 MOV A, #LINE_FEED...insert a line feed after the carriage return for QSC to transmit

0556 F0 = 1 1854 MOVX @DPTR, A...store byte in buffer

055A 5196 = 1 1859 CALL NEW_BUFFER1_IN...setup for next buffer if

055E 05FF = 1 1860 MOVX @DPTR, A...store byte in buffer

0562 5196 = 1 1861 CALL NEW_BUFFER1_IN...setup for next buffer if

0566 05FF = 1 1862...
.linefeed received

.2 needed for destination and
source address which do not
increment BYTE_COUNT when
loaded

continue outputting bytes
until counter reaches 0

clear active buffer bit for
last buffer used

.indicate that LSC is no longer
trying to transmit a packet

clear LSC transmit interrupt bit

load DPTR with address of
next byte to transmit

.get next byte

.load byte into LSC transmitter

.increment LSC input address

.store incremented address

.return to main program

.to get to this point means that a
message has been received which is
longer than the maximum specified
length - MAX_LENGTH (120)

SFRs to save before servicing
.interrupt
LOC  OBJ  LINE  SOURCE
0624  7BE1  =1  1918  MOV  ERROR_POINTER,  #LONG_COUNTER
0626  5175  =1  1919  CALL  INCREMENT_COUNTER
0628  8080  =1  1920  JMP  REC_ERROR_COUNT  END
062A  207109  =1  1921  CALL  INCR_INCREMENT,  COUNTER
062D  307003  =1  1922  CALL  INCR_INCREMENT,  COUNTER
0630  C277  =1  1923  CLR  ACTIVATE,  OUT
0632  22  =1  1924  RET
0633  C274  =1  1925  CLR  ACTIVATE,  OUT
0635  22  =1  1926  RET
0636  207003  =1  1927  CALL  INCLUDE,  (LSCHOT  SRC)
0639  C276  =1  1928  CALL  INCR_INCREMENT,  COUNTER
063C  C275  =1  1929  CLR  ACTIVATE,  OUT
063E  22  =1  1930  RET

LSC  OUT  LSC  OUT  LSC  OUT  LSC  OUT
Msb  =  1B.  buffer  just  emptied  must  be  2B  or  2C
Lsb  =  00B.  buffer  just  emptied  is  2D
Lsb  =  01B.  buffer  just  emptied  is  2A
Lsb  =  11B  then  buffer  just  emptied  must  be  2C
Lsb  =  10B  buffer  just  emptied  must  be  2B
Lsb  =  11B  LSC  OUT  =  11B
LSC  OUT  =  00B
LSC  OUT  =  01B
LSC  OUT  =  00B
LSC  OUT  =  11B
LSC  OUT  =  10B
LSC  OUT  =  01B

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE</th>
<th>ATTRIBUTES AND REFERENCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>NUMB</td>
<td>0006H</td>
<td>A</td>
</tr>
<tr>
<td>ACC</td>
<td>NUMB</td>
<td>000DH</td>
<td>A</td>
</tr>
<tr>
<td>ADDRESS_DETERMINATION</td>
<td>ADDR</td>
<td>0243H</td>
<td></td>
</tr>
<tr>
<td>ADRO</td>
<td>NUMB</td>
<td>0095H</td>
<td>A</td>
</tr>
<tr>
<td>ADR1</td>
<td>NUMB</td>
<td>00A5H</td>
<td>A</td>
</tr>
<tr>
<td>ADR2</td>
<td>NUMB</td>
<td>0085H</td>
<td>A</td>
</tr>
<tr>
<td>ADR3</td>
<td>NUMB</td>
<td>00C5H</td>
<td>A</td>
</tr>
<tr>
<td>AE_CHECK</td>
<td>ADDR</td>
<td>05A6H</td>
<td>A</td>
</tr>
<tr>
<td>AE_COUNTER</td>
<td>ADDR</td>
<td>00EDH</td>
<td>A</td>
</tr>
<tr>
<td>AE</td>
<td>NUMB</td>
<td>00EDH</td>
<td>A</td>
</tr>
<tr>
<td>AMSKO</td>
<td>NUMB</td>
<td>00D5H</td>
<td>A</td>
</tr>
<tr>
<td>AMSKI</td>
<td>NUMB</td>
<td>00E5H</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>NUMB</td>
<td>00F0H</td>
<td>A</td>
</tr>
<tr>
<td>BAUD</td>
<td>NUMB</td>
<td>0099H</td>
<td>A</td>
</tr>
<tr>
<td>BCRHO</td>
<td>NUMB</td>
<td>00E3H</td>
<td>A</td>
</tr>
<tr>
<td>BCRHI</td>
<td>NUMB</td>
<td>00F9H</td>
<td>A</td>
</tr>
<tr>
<td>BCRLO</td>
<td>NUMB</td>
<td>00EDH</td>
<td>A</td>
</tr>
<tr>
<td>BCRLI</td>
<td>NUMB</td>
<td>00F2H</td>
<td>A</td>
</tr>
<tr>
<td>BKF0F</td>
<td>NUMB</td>
<td>00C4H</td>
<td>A</td>
</tr>
<tr>
<td>BUF1A_ACTIVE</td>
<td>ADDR</td>
<td>02FH 4 A</td>
<td>297# 300 397 696 826 986 1479</td>
</tr>
<tr>
<td>BUF1A_ADDR</td>
<td>NUMB</td>
<td>0033H</td>
<td>A</td>
</tr>
<tr>
<td>BUF1B_ACTIVE</td>
<td>ADDR</td>
<td>02FH 5 A</td>
<td>294# 297 400 668 743 928 1495</td>
</tr>
<tr>
<td>BUF1B_ADDR</td>
<td>NUMB</td>
<td>0038H</td>
<td>A</td>
</tr>
<tr>
<td>BUF1C_ACTIVE</td>
<td>ADDR</td>
<td>02FH 6 A</td>
<td>291# 294 403 715 808 964 1502</td>
</tr>
<tr>
<td>BUF1C_ADDR</td>
<td>NUMB</td>
<td>0133H</td>
<td>A</td>
</tr>
<tr>
<td>BUF1D_ACTIVE</td>
<td>ADDR</td>
<td>02FH 7 A</td>
<td>280# 291 406 780 854 994 1471</td>
</tr>
<tr>
<td>BUF1D_ADDR</td>
<td>NUMB</td>
<td>0183H</td>
<td>A</td>
</tr>
<tr>
<td>BUF2A_ACTIVE</td>
<td>ADDR</td>
<td>02EH 7 A</td>
<td>316# 319 409 1102 1220 1293 1939</td>
</tr>
<tr>
<td>BUF2A_ADDR</td>
<td>NUMB</td>
<td>0201H</td>
<td>A</td>
</tr>
<tr>
<td>BUF2B_ACTIVE</td>
<td>ADDR</td>
<td>02EH 6 A</td>
<td>319# 322 412 1080 1143 1222 1957</td>
</tr>
<tr>
<td>BUF2B_ADDR</td>
<td>NUMB</td>
<td>0261H</td>
<td>A</td>
</tr>
<tr>
<td>BUF2C_ACTIVE</td>
<td>ADDR</td>
<td>02EH 5 A</td>
<td>322# 325 415 1121 1202 1357 1964</td>
</tr>
<tr>
<td>BUF2C_ADDR</td>
<td>NUMB</td>
<td>0301H</td>
<td>A</td>
</tr>
<tr>
<td>BUF2D_ACTIVE</td>
<td>ADDR</td>
<td>02EH 4 A</td>
<td>325# 328 418 1180 1242 1386 1946</td>
</tr>
<tr>
<td>BUF2D_ADDR</td>
<td>NUMB</td>
<td>03B1H</td>
<td>A</td>
</tr>
<tr>
<td>BUFFER1_CONTROL</td>
<td>ADDR</td>
<td>02FH A</td>
<td>280# 530 1580</td>
</tr>
<tr>
<td>BUFFER1_ADDR</td>
<td>ADDR</td>
<td>012DH</td>
<td>A</td>
</tr>
<tr>
<td>BUFFER1B_ADDR</td>
<td>ADDR</td>
<td>012CH</td>
<td>A</td>
</tr>
<tr>
<td>BUFFER1_RELOAD</td>
<td>ADDR</td>
<td>051BH</td>
<td>A</td>
</tr>
<tr>
<td>BUFFER1D_ADDR</td>
<td>ADDR</td>
<td>0530H</td>
<td>A</td>
</tr>
<tr>
<td>BUFFER1D_RELOAD</td>
<td>ADDR</td>
<td>053AH</td>
<td>A</td>
</tr>
<tr>
<td>BUFFER2_CONTROL</td>
<td>ADDR</td>
<td>002EH</td>
<td>A</td>
</tr>
<tr>
<td>BUFFER2_ADDR</td>
<td>ADDR</td>
<td>0131H</td>
<td>A</td>
</tr>
<tr>
<td>CLEAR_ACTIVE_IA</td>
<td>ADDR</td>
<td>04C4H</td>
<td>A</td>
</tr>
<tr>
<td>CLEAR_ACTIVE_IB_1C</td>
<td>ADDR</td>
<td>04C6H</td>
<td>A</td>
</tr>
<tr>
<td>CLEAR_ACTIVE_IB</td>
<td>ADDR</td>
<td>04CEH</td>
<td>A</td>
</tr>
<tr>
<td>CLEAR_ACTIVE_IC</td>
<td>ADDR</td>
<td>04D3H</td>
<td>A</td>
</tr>
<tr>
<td>CLEAR_ACTIVE_ID</td>
<td>ADDR</td>
<td>04BCH</td>
<td>A</td>
</tr>
<tr>
<td>CLEAR_ACTIVE_BUFFER</td>
<td>ADDR</td>
<td>04B9H</td>
<td>A</td>
</tr>
<tr>
<td>CLR_ACT_2A_2D</td>
<td>ADDR</td>
<td>062DH</td>
<td>A</td>
</tr>
<tr>
<td>NAME</td>
<td>TYPE</td>
<td>VALUE</td>
<td>ATTRIBUTES AND REFERENCES</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
<td>-------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>CLR_ACT_2A</td>
<td>C ADDR</td>
<td>0630H A</td>
<td>1937#</td>
</tr>
<tr>
<td>CLR_ACT_2B, 2C</td>
<td>C ADDR</td>
<td>0636H A</td>
<td>1929 1950#</td>
</tr>
<tr>
<td>CLR_ACT_2B</td>
<td>C ADDR</td>
<td>0639H A</td>
<td>1955#</td>
</tr>
<tr>
<td>CLR_ACT_2C</td>
<td>C ADDR</td>
<td>063CH A</td>
<td>1992 1962#</td>
</tr>
<tr>
<td>CLR_ACT_2D</td>
<td>C ADDR</td>
<td>0639H A</td>
<td>1934 1944#</td>
</tr>
<tr>
<td>CLR_ACTIVE_OUT</td>
<td>C ADDR</td>
<td>062AH A</td>
<td>1879 1927#</td>
</tr>
<tr>
<td>COUNTER_CLEAR</td>
<td>C ADDR</td>
<td>026EH A</td>
<td>558 559</td>
</tr>
<tr>
<td>COUNTER_OVERFLOW</td>
<td>C ADDR</td>
<td>0282H A</td>
<td>583 594#</td>
</tr>
<tr>
<td>CR</td>
<td>NUMB</td>
<td>000DH A</td>
<td>204# 1848</td>
</tr>
<tr>
<td>CRC_CHECK</td>
<td>C ADDR</td>
<td>059CH A</td>
<td>1754 1762#</td>
</tr>
<tr>
<td>CRCE_COUNTER</td>
<td>D ADDR</td>
<td>00E7H A</td>
<td>232# 235 1765</td>
</tr>
<tr>
<td>CRCE</td>
<td>NUMB</td>
<td>00ECH A</td>
<td>159# 1763</td>
</tr>
<tr>
<td>CY</td>
<td>NUMB</td>
<td>00D7H A</td>
<td>75#</td>
</tr>
<tr>
<td>DARHO</td>
<td>NUMB</td>
<td>00C3H A</td>
<td>50#</td>
</tr>
<tr>
<td>DARH1</td>
<td>NUMB</td>
<td>00D3H A</td>
<td>55# 476 1259</td>
</tr>
<tr>
<td>DARL0</td>
<td>NUMB</td>
<td>00C2H A</td>
<td>49# 453</td>
</tr>
<tr>
<td>DARL1</td>
<td>NUMB</td>
<td>00D2H A</td>
<td>54# 475 1258</td>
</tr>
<tr>
<td>DCONO</td>
<td>NUMB</td>
<td>0092H A</td>
<td>36# 455 1028 1536 1678</td>
</tr>
<tr>
<td>DCONI</td>
<td>NUMB</td>
<td>0093H A</td>
<td>37# 464 1701 1783</td>
</tr>
<tr>
<td>DMA</td>
<td>NUMB</td>
<td>00B8H A</td>
<td>153# 451</td>
</tr>
<tr>
<td>DMA1_DONE</td>
<td>C ADDR</td>
<td>0053H A</td>
<td>375#</td>
</tr>
<tr>
<td>DMA1_SERVICE</td>
<td>C ADDR</td>
<td>001CH A</td>
<td>376# 1909#</td>
</tr>
<tr>
<td>DPH</td>
<td>NUMB</td>
<td>0083H A</td>
<td>19# 673 720 785 831 916 946 982 1012 1085 1126 1185 1225 1422</td>
</tr>
<tr>
<td>DPL</td>
<td>NUMB</td>
<td>0082H A</td>
<td>1438 1524 1539 1682 1691 1707 1715 1789 1798 1810 1821 1835 1846</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1894 1904 1914</td>
</tr>
<tr>
<td>EA</td>
<td>NUMB</td>
<td>00AFH A</td>
<td>94# 523</td>
</tr>
<tr>
<td>EDMO</td>
<td>NUMB</td>
<td>00CAH A</td>
<td>133#</td>
</tr>
<tr>
<td>EDM1</td>
<td>NUMB</td>
<td>00CCH A</td>
<td>1314 521</td>
</tr>
<tr>
<td>EDSRE</td>
<td>NUMB</td>
<td>00C9H A</td>
<td>134# 517</td>
</tr>
<tr>
<td>EDSRV</td>
<td>NUMB</td>
<td>00CBH A</td>
<td>135# 515</td>
</tr>
<tr>
<td>EGSTE</td>
<td>NUMB</td>
<td>00CDH A</td>
<td>1308 1026 1449</td>
</tr>
<tr>
<td>EGSTV</td>
<td>NUMB</td>
<td>00CBH A</td>
<td>1328 1023 1447</td>
</tr>
<tr>
<td>END_CLEAR_ACTIVE_OUT</td>
<td>C ADDR</td>
<td>04D5H A</td>
<td>146A 1475 1486 1498 1507#</td>
</tr>
<tr>
<td>END_LSC_RECEIVE</td>
<td>C ADDR</td>
<td>05FEH A</td>
<td>1848 1870#</td>
</tr>
<tr>
<td>ERROR_POINTER</td>
<td>REG</td>
<td>RO</td>
<td>2098 577 577 599 596 598 600 602 604 606 608 610 612 614 616 618</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1549 1559 1566 1747 1756 1765 1772 1919</td>
</tr>
<tr>
<td>ES</td>
<td>NUMB</td>
<td>00ACH A</td>
<td>95# 519</td>
</tr>
<tr>
<td>ET0</td>
<td>NUMB</td>
<td>00AH A</td>
<td>90#</td>
</tr>
<tr>
<td>ET1</td>
<td>NUMB</td>
<td>00ABH A</td>
<td>96#</td>
</tr>
<tr>
<td>EX0</td>
<td>NUMB</td>
<td>00ABH A</td>
<td>99#</td>
</tr>
<tr>
<td>EX1</td>
<td>NUMB</td>
<td>00A9H A</td>
<td>97#</td>
</tr>
<tr>
<td>FO</td>
<td>NUMB</td>
<td>00D9H A</td>
<td>77#</td>
</tr>
<tr>
<td>FIRST_GSC_OUT</td>
<td>B ADDR</td>
<td>002DH 7 A</td>
<td>344# 346 481 1464 1482</td>
</tr>
<tr>
<td>GENERIC_INIT</td>
<td>C ADDR</td>
<td>025BH A</td>
<td>370# 520#</td>
</tr>
<tr>
<td>GMD0</td>
<td>NUMB</td>
<td>0084H A</td>
<td>34# 444</td>
</tr>
<tr>
<td>GREN</td>
<td>NUMB</td>
<td>00E9H A</td>
<td>162A 479 1703 1785</td>
</tr>
<tr>
<td>GSC_BAUD_RATE</td>
<td>C ADDR</td>
<td>000DH A</td>
<td>166A 442</td>
</tr>
<tr>
<td>GSC_DEST_ADDR</td>
<td>D ADDR</td>
<td>0077H A</td>
<td>257# 260 508 685 732 797 843</td>
</tr>
<tr>
<td>GSC_ERROR_REC</td>
<td>C ADDR</td>
<td>05B0H A</td>
<td>364 1712#</td>
</tr>
<tr>
<td>GSC_ERROR_XMIT_END</td>
<td>C ADDR</td>
<td>0500H A</td>
<td>1552 1562 1560#</td>
</tr>
<tr>
<td>GSC_ERROR_XMIT</td>
<td>C ADDR</td>
<td>04E3H A</td>
<td>372 1503#</td>
</tr>
<tr>
<td>GSC_IN_2A</td>
<td>C ADDR</td>
<td>0417H A</td>
<td>1175 1210#</td>
</tr>
<tr>
<td>NAME</td>
<td>TYPE</td>
<td>VALUE</td>
<td>ATTRIBUTES AND REFERENCES</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
<td>-------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>GSC_IN_2B</td>
<td>C</td>
<td>ADDR 03A4H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_IN_2C</td>
<td>C</td>
<td>ADDR 03A4H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_IN_2D</td>
<td>C</td>
<td>ADDR 03A4H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_IN_LSB</td>
<td>B</td>
<td>ADDR 002EH.2</td>
<td>A</td>
</tr>
<tr>
<td>GSC_IN_MSB</td>
<td>B</td>
<td>ADDR 002EH.3</td>
<td>A</td>
</tr>
<tr>
<td>GSC_INIT</td>
<td>C</td>
<td>ADDR 0200H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_INPUT_HIGH</td>
<td>D</td>
<td>ADDR 007EH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_INPUT_LOW</td>
<td>D</td>
<td>ADDR 007EH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_OUT_IA</td>
<td>C</td>
<td>ADDR 033EH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_OUT_IB</td>
<td>C</td>
<td>ADDR 035EH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_OUT_IC_1D</td>
<td>C</td>
<td>ADDR 037EH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_OUT_IC</td>
<td>C</td>
<td>ADDR 0375H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_OUT_10</td>
<td>C</td>
<td>ADDR 008FH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_OUT_LSB</td>
<td>B</td>
<td>ADDR 002FH.2</td>
<td>A</td>
</tr>
<tr>
<td>GSC_OUT_MSB</td>
<td>B</td>
<td>ADDR 002FH.3</td>
<td>A</td>
</tr>
<tr>
<td>GSC_REC_ERROR</td>
<td>C</td>
<td>ADDR 0023H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_REC_INVALID</td>
<td>C</td>
<td>ADDR 0023H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_SRC_ADDR</td>
<td>D</td>
<td>ADDR 007CH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_VALID_RECV</td>
<td>C</td>
<td>ADDR 0054H</td>
<td>A</td>
</tr>
<tr>
<td>GSC_VALID_XMIT</td>
<td>C</td>
<td>ADDR 04AAH</td>
<td>A</td>
</tr>
<tr>
<td>GSC_XMIT_ERROR</td>
<td>C</td>
<td>ADDR 0043H</td>
<td>A</td>
</tr>
<tr>
<td>HABEN</td>
<td>NUMB</td>
<td>ADDR 00E8H</td>
<td>A</td>
</tr>
<tr>
<td>IE</td>
<td>NUMB</td>
<td>ADDR 00A0H</td>
<td>A</td>
</tr>
<tr>
<td>IEI</td>
<td>NUMB</td>
<td>ADDR 00B8H</td>
<td>A</td>
</tr>
<tr>
<td>IENI</td>
<td>NUMB</td>
<td>ADDR 00C8H</td>
<td>A</td>
</tr>
<tr>
<td>IFS_PERIOD</td>
<td>NUMB</td>
<td>ADDR 0014H</td>
<td>A</td>
</tr>
<tr>
<td>IFS</td>
<td>NUMB</td>
<td>ADDR 00A4H</td>
<td>A</td>
</tr>
<tr>
<td>IN_BYTE_COUNT</td>
<td>D</td>
<td>ADDR 007FH</td>
<td>A</td>
</tr>
<tr>
<td>INC_COUNT_LOOP</td>
<td>C</td>
<td>ADDR 027EH</td>
<td>A</td>
</tr>
<tr>
<td>INC_ERROR_COUNT</td>
<td>C</td>
<td>ADDR 05BH</td>
<td>A</td>
</tr>
<tr>
<td>INCREMENT_COUNTER</td>
<td>C</td>
<td>ADDR 0275H</td>
<td>A</td>
</tr>
<tr>
<td>INITIALIZATION</td>
<td>C</td>
<td>ADDR 0100H</td>
<td>A</td>
</tr>
<tr>
<td>INTO</td>
<td>NUMB</td>
<td>ADDR 0022H</td>
<td>A</td>
</tr>
<tr>
<td>INTI</td>
<td>NUMB</td>
<td>ADDR 0022H</td>
<td>A</td>
</tr>
<tr>
<td>INTERRUPT_ENABLE</td>
<td>C</td>
<td>ADDR 0250H</td>
<td>A</td>
</tr>
<tr>
<td>IP</td>
<td>NUMB</td>
<td>ADDR 008BH</td>
<td>A</td>
</tr>
<tr>
<td>IPI</td>
<td>NUMB</td>
<td>ADDR 00F8H</td>
<td>A</td>
</tr>
<tr>
<td>IRET</td>
<td>C</td>
<td>ADDR 032EH</td>
<td>A</td>
</tr>
<tr>
<td>ITO</td>
<td>NUMB</td>
<td>ADDR 008BH</td>
<td>A</td>
</tr>
<tr>
<td>ITI</td>
<td>NUMB</td>
<td>ADDR 008AH</td>
<td>A</td>
</tr>
<tr>
<td>LINE_FEED</td>
<td>NUMB</td>
<td>ADDR 000AH</td>
<td>A</td>
</tr>
<tr>
<td>LNI</td>
<td>NUMB</td>
<td>ADDR 00D8H</td>
<td>A</td>
</tr>
<tr>
<td>LONG_COUNTER</td>
<td>D</td>
<td>ADDR 00E1H</td>
<td>A</td>
</tr>
<tr>
<td>LSC_ACTIVE</td>
<td>B</td>
<td>ADDR 002DH.6</td>
<td>A</td>
</tr>
<tr>
<td>LSC_BAUD_RATE</td>
<td>NUMB</td>
<td>ADDR 00FCH</td>
<td>A</td>
</tr>
<tr>
<td>LSC_IN_1A</td>
<td>C</td>
<td>ADDR 03D0H</td>
<td>A</td>
</tr>
<tr>
<td>LSC_IN_1B</td>
<td>C</td>
<td>ADDR 029CH</td>
<td>A</td>
</tr>
<tr>
<td>LSC_IN_1C</td>
<td>C</td>
<td>ADDR 028FH</td>
<td>A</td>
</tr>
<tr>
<td>LSC_IN_1D</td>
<td>C</td>
<td>ADDR 0275H</td>
<td>A</td>
</tr>
<tr>
<td>LSC_IN_2A</td>
<td>C</td>
<td>ADDR 0275H</td>
<td>A</td>
</tr>
<tr>
<td>LSC_IN_2B</td>
<td>B</td>
<td>ADDR 002FH.1</td>
<td>A</td>
</tr>
<tr>
<td>NAME</td>
<td>TYPE</td>
<td>VALUE</td>
<td>ATTRIBUTES AND REFERENCES</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
<td>--------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>LSC_INIT</td>
<td>C</td>
<td>0234H</td>
<td>A 388 486#</td>
</tr>
<tr>
<td>LSC_INPUT_HIGH</td>
<td>D</td>
<td>007AH</td>
<td>A 264 268 543 706 753 818 864 1835 1846</td>
</tr>
<tr>
<td>LSC_INPUT_LOW</td>
<td>D</td>
<td>007BH</td>
<td>A 263 264 542 705 752 817 863 1834 1845</td>
</tr>
<tr>
<td>LSC_OUT_2A</td>
<td>C</td>
<td>044EH</td>
<td>A 1290#</td>
</tr>
<tr>
<td>LSC_OUT_2B</td>
<td>C</td>
<td>0462H</td>
<td>A 1287 1319#</td>
</tr>
<tr>
<td>LSC_OUT_2C_2D</td>
<td>C</td>
<td>0476H</td>
<td>A 1284 1342 1378 1407 1924 1952</td>
</tr>
<tr>
<td>LSC_OUT_2C</td>
<td>C</td>
<td>0479H</td>
<td>A 1354#</td>
</tr>
<tr>
<td>LSC_OUT_2D</td>
<td>C</td>
<td>048DH</td>
<td>A 1350 1383#</td>
</tr>
<tr>
<td>LSC_OUT_COUNTER</td>
<td>D</td>
<td>0075H</td>
<td>A 277 288 305 334 342 369 398 420 1876 1877</td>
</tr>
<tr>
<td>LSC_INPUT_HIGH</td>
<td>D</td>
<td>0076H</td>
<td>A 274 277 302 342 377 404 1894 1904 1905</td>
</tr>
<tr>
<td>LSC_INPUT_LOW</td>
<td>D</td>
<td>0077H</td>
<td>A 273 274 301 342 377 403 1893 1903 1904</td>
</tr>
<tr>
<td>LSC_OUT_HIGH</td>
<td>D</td>
<td>0078H</td>
<td>A 213 241 461 591 632 791 1132 1191 1231 1264</td>
</tr>
<tr>
<td>LSC_OUTPUT_LOW</td>
<td>D</td>
<td>0079H</td>
<td>A 213 241 461 591 632 791 1132 1191 1231 1264</td>
</tr>
<tr>
<td>LSC_RECEIVE</td>
<td>C</td>
<td>0500H</td>
<td>A 1806 1826#</td>
</tr>
<tr>
<td>LSC_SERVICE</td>
<td>C</td>
<td>058AH</td>
<td>A 356 1795#</td>
</tr>
<tr>
<td>LSC_XMIT</td>
<td>C</td>
<td>05F2H</td>
<td>A 1817 1874#</td>
</tr>
<tr>
<td>MAIN</td>
<td>C</td>
<td>0112H</td>
<td>A 395 421 428 436 501 513 540 552 572 594 639</td>
</tr>
<tr>
<td>MAX_LENGTH</td>
<td>NUM</td>
<td>0078H</td>
<td>A 213 461 1091 1132 1191 1231 1264</td>
</tr>
<tr>
<td>MSGLOT</td>
<td>NUM</td>
<td>00F5H</td>
<td>A 678#</td>
</tr>
<tr>
<td>NEW_BUF1_IN_END</td>
<td>C</td>
<td>03CDH</td>
<td>A 710 757 822 868#</td>
</tr>
<tr>
<td>NEW_BUF2_IN_END</td>
<td>C</td>
<td>0432H</td>
<td>A 1116 1157 1216 1256#</td>
</tr>
<tr>
<td>NEW_BUFFER1_IN</td>
<td>C</td>
<td>0296H</td>
<td>A 624 770 1862#</td>
</tr>
<tr>
<td>NEW_BUFFER1_OUT</td>
<td>C</td>
<td>032FH</td>
<td>A 422 975 1514#</td>
</tr>
<tr>
<td>NEW_BUFFER2_IN</td>
<td>C</td>
<td>03B0H</td>
<td>A 1036 1170 1696 1781#</td>
</tr>
<tr>
<td>NEW_BUFFER2_OUT</td>
<td>C</td>
<td>043FH</td>
<td>A 432 1267#</td>
</tr>
<tr>
<td>NEXT_LOCATION</td>
<td>D</td>
<td>00DFH</td>
<td>A 243 352#</td>
</tr>
<tr>
<td>NOACK_COUNTER</td>
<td>C</td>
<td>00D5H</td>
<td>A 18 1906#</td>
</tr>
<tr>
<td>NOACK_ERROR</td>
<td>C</td>
<td>04F6H</td>
<td>A 1546 1554#</td>
</tr>
<tr>
<td>NOTHING_FOR_GSC</td>
<td>C</td>
<td>03AFH</td>
<td>A 882 899 928 964 994 1303#</td>
</tr>
<tr>
<td>NOTHING_FOR_LSC</td>
<td>C</td>
<td>04AFH</td>
<td>A 1277 1293 1322 1357 1386 1429#</td>
</tr>
<tr>
<td>OUT_BYTE_COUNT</td>
<td>D</td>
<td>007EH</td>
<td>A 253 257#</td>
</tr>
<tr>
<td>OV</td>
<td>NUM</td>
<td>00D2H</td>
<td>A 80#</td>
</tr>
<tr>
<td>OVR_CHECK</td>
<td>C</td>
<td>0592H</td>
<td>A 1745 1753#</td>
</tr>
<tr>
<td>OVR_COUNTER</td>
<td>D</td>
<td>00F2H</td>
<td>A 2232 226 1756#</td>
</tr>
<tr>
<td>OVR</td>
<td>NUM</td>
<td>00EFH</td>
<td>A 156# 1754#</td>
</tr>
<tr>
<td>P</td>
<td>NUM</td>
<td>00D0H</td>
<td>A 81#</td>
</tr>
<tr>
<td>P0</td>
<td>NUM</td>
<td>00D0H</td>
<td>A 10#</td>
</tr>
<tr>
<td>P1</td>
<td>NUM</td>
<td>00D0H</td>
<td>A 11# 301 506#</td>
</tr>
<tr>
<td>P2</td>
<td>NUM</td>
<td>00A0H</td>
<td>A 12#</td>
</tr>
<tr>
<td>P3</td>
<td>NUM</td>
<td>00B0H</td>
<td>A 13#</td>
</tr>
<tr>
<td>P4</td>
<td>NUM</td>
<td>00C0H</td>
<td>A 48# 503 50B#</td>
</tr>
<tr>
<td>PCON</td>
<td>NUM</td>
<td>00B7H</td>
<td>A 20#</td>
</tr>
<tr>
<td>PDMAD</td>
<td>NUM</td>
<td>00F4H</td>
<td>A 141#</td>
</tr>
<tr>
<td>PDMAI</td>
<td>NUM</td>
<td>00FCH</td>
<td>A 139#</td>
</tr>
<tr>
<td>PGSE</td>
<td>NUM</td>
<td>00F9H</td>
<td>A 142#</td>
</tr>
<tr>
<td>PGSRV</td>
<td>NUM</td>
<td>00F8H</td>
<td>A 143#</td>
</tr>
<tr>
<td>PGSTE</td>
<td>NUM</td>
<td>00F9H</td>
<td>A 138#</td>
</tr>
<tr>
<td>PGTV</td>
<td>NUM</td>
<td>00F8H</td>
<td>A 140#</td>
</tr>
<tr>
<td>PRBS</td>
<td>NUM</td>
<td>00E4H</td>
<td>A 61#</td>
</tr>
<tr>
<td>PS</td>
<td>NUM</td>
<td>00BEH</td>
<td>A 102#</td>
</tr>
<tr>
<td>PSW</td>
<td>NUM</td>
<td>00D0H</td>
<td>A 148 1440 1522 1541 1680 1693 1705 1717 1724 1737 1787 1800 1808 1819 1916</td>
</tr>
<tr>
<td>NAME</td>
<td>TYPE</td>
<td>VALUE</td>
<td>ATTRIBUTES AND REFERENCES</td>
</tr>
<tr>
<td>----------------------</td>
<td>------</td>
<td>---------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>PTO</td>
<td>NUMB</td>
<td>0089H</td>
<td>A 105#</td>
</tr>
<tr>
<td>PT1</td>
<td>NUMB</td>
<td>008BH</td>
<td>A 106#</td>
</tr>
<tr>
<td>PX1</td>
<td>NUMB</td>
<td>008AH</td>
<td>A 107#</td>
</tr>
<tr>
<td>RBB</td>
<td>NUMB</td>
<td>009AH</td>
<td>A 124#</td>
</tr>
<tr>
<td>RCABT_CHECK</td>
<td>C</td>
<td>ADDR</td>
<td>0588H</td>
</tr>
<tr>
<td>RCABT_COUNTER</td>
<td>D</td>
<td>ADDR</td>
<td>00F3H</td>
</tr>
<tr>
<td>RCABT</td>
<td>NUMB</td>
<td>00E3H</td>
<td>A 157# 1743</td>
</tr>
<tr>
<td>RD</td>
<td>NUMB</td>
<td>0087H</td>
<td>A 108#</td>
</tr>
<tr>
<td>RDN</td>
<td>NUMB</td>
<td>00E8H</td>
<td>A 160#</td>
</tr>
<tr>
<td>REC_ERROR_COUNT_END</td>
<td>C</td>
<td>ADDR</td>
<td>05AAH</td>
</tr>
<tr>
<td>REN</td>
<td>NUMB</td>
<td>0090H</td>
<td>A 122#</td>
</tr>
<tr>
<td>RFIFO</td>
<td>NUMB</td>
<td>00F4H</td>
<td>A 166# 45B</td>
</tr>
<tr>
<td>RFNE</td>
<td>NUMB</td>
<td>00E7H</td>
<td>A 167#</td>
</tr>
<tr>
<td>RI</td>
<td>NUMB</td>
<td>009BH</td>
<td>A 126# 1803 1828</td>
</tr>
<tr>
<td>RS0</td>
<td>NUMB</td>
<td>00D3H</td>
<td>A 168#</td>
</tr>
<tr>
<td>RS1</td>
<td>NUMB</td>
<td>00D4H</td>
<td>A 169#</td>
</tr>
<tr>
<td>RSTAT</td>
<td>NUMB</td>
<td>00E8H</td>
<td>A 170#</td>
</tr>
<tr>
<td>RXD</td>
<td>NUMB</td>
<td>0080H</td>
<td>A 171#</td>
</tr>
<tr>
<td>SAVHO</td>
<td>NUMB</td>
<td>00A3H</td>
<td>A 172# 916 945 982 1012 1589 1610 1631 1649</td>
</tr>
<tr>
<td>SAVHI</td>
<td>NUMB</td>
<td>00B3H</td>
<td>A 173#</td>
</tr>
<tr>
<td>SARLO</td>
<td>NUMB</td>
<td>00A2H</td>
<td>A 174# 915 945 981 1011 1588 1609 1630 1648</td>
</tr>
<tr>
<td>SARLI</td>
<td>NUMB</td>
<td>00B2H</td>
<td>A 175#</td>
</tr>
<tr>
<td>SBUF</td>
<td>NUMB</td>
<td>0099H</td>
<td>A 176#</td>
</tr>
<tr>
<td>SCON</td>
<td>NUMB</td>
<td>0098H</td>
<td>A 177# 29# 492</td>
</tr>
<tr>
<td>SECOND_LSC_CHECK</td>
<td>C</td>
<td>ADDR</td>
<td>0033H</td>
</tr>
<tr>
<td>SECOND_TEN_CHECK</td>
<td>C</td>
<td>ADDR</td>
<td>0034H</td>
</tr>
<tr>
<td>SLOTTM</td>
<td>NUMB</td>
<td>0084H</td>
<td>A 178#</td>
</tr>
<tr>
<td>SM0</td>
<td>NUMB</td>
<td>00F9H</td>
<td>A 179#</td>
</tr>
<tr>
<td>SM1</td>
<td>NUMB</td>
<td>00F8H</td>
<td>A 180#</td>
</tr>
<tr>
<td>SM2</td>
<td>NUMB</td>
<td>00D2H</td>
<td>A 181#</td>
</tr>
<tr>
<td>SP</td>
<td>NUMB</td>
<td>0081H</td>
<td>A 182# 38#</td>
</tr>
<tr>
<td>STACK_OFFSET</td>
<td>NUMB</td>
<td>00B0H</td>
<td>A 183# 202# 38#</td>
</tr>
<tr>
<td>START_GSC_OUT</td>
<td>C</td>
<td>ADDR</td>
<td>03AAH</td>
</tr>
<tr>
<td>START_LSC_OUT</td>
<td>C</td>
<td>ADDR</td>
<td>049EH</td>
</tr>
<tr>
<td>START_RETRANSMIT</td>
<td>C</td>
<td>ADDR</td>
<td>0554H</td>
</tr>
<tr>
<td>START</td>
<td>C</td>
<td>ADDR</td>
<td>0000H</td>
</tr>
<tr>
<td>T0</td>
<td>NUMB</td>
<td>00B4H</td>
<td>A 186#</td>
</tr>
<tr>
<td>T1</td>
<td>NUMB</td>
<td>00B5H</td>
<td>A 187#</td>
</tr>
<tr>
<td>TBB</td>
<td>NUMB</td>
<td>00B6H</td>
<td>A 188#</td>
</tr>
<tr>
<td>TCDCNT</td>
<td>NUMB</td>
<td>00D4H</td>
<td>A 189# 56# 449 1520 1668</td>
</tr>
<tr>
<td>TCDD_COUNTER</td>
<td>D</td>
<td>ADDR</td>
<td>00DBH</td>
</tr>
<tr>
<td>TCDD_ERROR</td>
<td>C</td>
<td>ADDR</td>
<td>04F2H</td>
</tr>
<tr>
<td>TCDT</td>
<td>NUMB</td>
<td>00CH</td>
<td>A 190#</td>
</tr>
<tr>
<td>TCON</td>
<td>NUMB</td>
<td>00B8H</td>
<td>A 191#</td>
</tr>
<tr>
<td>TDN</td>
<td>NUMB</td>
<td>00B9H</td>
<td>A 192#</td>
</tr>
<tr>
<td>TEN</td>
<td>NUMB</td>
<td>00D9H</td>
<td>A 193# 877 886 1021 1670 1672</td>
</tr>
<tr>
<td>TFO</td>
<td>NUMB</td>
<td>00BDH</td>
<td>A 194#</td>
</tr>
<tr>
<td>TFI</td>
<td>NUMB</td>
<td>00BFH</td>
<td>A 195#</td>
</tr>
<tr>
<td>TFIFO</td>
<td>NUMB</td>
<td>00B5H</td>
<td>A 196# 35# 453</td>
</tr>
<tr>
<td>TFNF</td>
<td>NUMB</td>
<td>00DAH</td>
<td>A 197#</td>
</tr>
<tr>
<td>TGO</td>
<td>NUMB</td>
<td>00BCH</td>
<td>A 198#</td>
</tr>
<tr>
<td>THI</td>
<td>NUMB</td>
<td>00B8H</td>
<td>A 199# 26# 497</td>
</tr>
<tr>
<td>TI</td>
<td>NUMB</td>
<td>0099H</td>
<td>A 200# 125# 1425 1887</td>
</tr>
<tr>
<td>TLO</td>
<td>NUMB</td>
<td>008AH</td>
<td>A 201#</td>
</tr>
</tbody>
</table>
### NAME  TYPE  VALUE  ATTRIBUTES AND REFERENCES

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>VALUE</th>
<th>Attributes and References</th>
</tr>
</thead>
<tbody>
<tr>
<td>TL1</td>
<td>NUMB</td>
<td>008BH</td>
<td>A 240</td>
</tr>
<tr>
<td>TMD</td>
<td>NUMB</td>
<td>008FH</td>
<td>A 228 489 490</td>
</tr>
<tr>
<td>TRO</td>
<td>NUMB</td>
<td>008CH</td>
<td>A 870</td>
</tr>
<tr>
<td>TR1</td>
<td>NUMB</td>
<td>008EH</td>
<td>A 850 495</td>
</tr>
<tr>
<td>TRANSMISSION_IN_PROGRESS</td>
<td>C</td>
<td>ADDR</td>
<td>0332H A 8810 886</td>
</tr>
<tr>
<td>TSTAT</td>
<td>NUMB</td>
<td>ADDR</td>
<td>008BH A 3000</td>
</tr>
<tr>
<td>TXD</td>
<td>NUMB</td>
<td>ADDR</td>
<td>0081H A 1158</td>
</tr>
<tr>
<td>UR_COUNTER</td>
<td>D</td>
<td>ADDR</td>
<td>00FFH A 2190 223 1549</td>
</tr>
<tr>
<td>UR_ERROR</td>
<td>C</td>
<td>ADDR</td>
<td>01EH A 1544</td>
</tr>
<tr>
<td>UR</td>
<td>NUMB</td>
<td>ADDR</td>
<td>00DDH A 1400 1546</td>
</tr>
<tr>
<td>WR</td>
<td>NUMB</td>
<td>ADDR</td>
<td>00FH A 1100</td>
</tr>
<tr>
<td>XMIT_LSC</td>
<td>C</td>
<td>ADDR</td>
<td>0501H A 1803 1815</td>
</tr>
</tbody>
</table>

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
APPENDIX B
TAKING CONTROL OF THE BACKOFF ALGORITHM

There is a method that allows the user to take control of the backoff process. This method will only work when normal or alternate backoff modes are selected. It will not work in DCR mode. This method works by loading TCDCNT with 80H. Then on the first collision, TCDCNT will overflow, aborting the transmission and causing a transmission error to occur. It is in the error routine where the user takes control. Some of the modifications that have been tested are:

1) Extending the number of retransmissions—this was accomplished by counting the number of attempted transmissions in a user implemented counter. When the number of collisions grew too big, the transmissions were aborted and an error flag set.

2) Extending the number of time slots available—to implement this, it was required that the time slots be simulated using one of the timers. Then by reading the PRBS multiple times and ANDing each read of the PRBS with a masking register, the number of time slots could be extended to randomly fall within any range selected by the user. Once the slot time was determined, the resulting value was multiplied by the selected time slot with the appropriate value loaded into the timer registers and the timer started. When the timer expired, the transmission was re-attempted. For very large delays, multiple timer overflows were required and a loop counter used. This also allowed time slots larger than 255 bit times to be used.

Other modifications the user may wish to implement would be to use some kind of token passing scheme when collisions occur or instead of randomly assigning slot times, assign pre-determined time slots to each station.

If the user decides to implement some kind of scheme such as these there are several factors the user must be aware of. These are:

1) When TCDCNT overflows, it will still contain either 0 or 1 and these many time slots must expire before the GSC will begin transmissions again. Even if the transmitter is disabled and re-enabled the GSC still goes through the standard backoff algorithm. This means the user should program the slot time to 01 to minimize the amount of time until the GSC hardware will allow another transmission to begin.

2) Due to the amount of software required to implement any of these suggestions, most will not work at the same speed the internal hardware is capable of. For this reason, running at maximum baud rates with minimum IFS will probably not work.

3) There is no real time indication to the user that the GSC thinks it is in a backoff algorithm, if the GSC is currently receiving data, or when a collision is detected. These, and possibly other factors not apparent at the time this application note was written, must be considered whenever the user tries to modify the hardware based backoff algorithm with software.
APPENDIX C
REFERENCES

Using the 8051 Microcontroller with Resonant Transducers

TOM WILLIAMSON

Abstract—Having to interface an analog transducer to a digital control system through an analog-to-digital converter represents an expensive bottleneck in the development of many systems. Some transducer companies are addressing this problem by developing proprietary families of resonant transducers.

Resonant transducers are oscillators whose frequency depends in some known way on the physical property being measured. The electrical output from these devices is a train of rectangular pulses whose repetition rate encodes the value of the measurand. Changes in the measurand cause the frequency to shift. The microcontroller detects the frequency shift, converts it in software to a measure of the measurand value.

This paper discusses software interfacing techniques between resonant transducers and the 8051. Techniques for measuring frequency and period are discussed and compared for resolution and interpolation times. The 8051 is capable of performing these tasks in extremely short CPU time. Requirements for obtaining n-bit resolution in the measurement are discussed. It is determined that it is always faster to evaluate the measurand to a given level of resolution by measuring the period rather than the frequency, even if the measurand is proportional to the frequency rather than to the period. Numerical and software examples are presented to illustrate the concepts.

1. Resonant Transducers

Most sensing transducers are not directly compatible with digital controllers, because they generate analog signals. A few transducer companies are developing proprietary families of sensors which generate signals that are more directly compatible with digital systems. These are not analog sensors with built-in A-D conversion, but oscillators whose frequency depends in some known way on the physical property being measured.

The technology is applicable to virtually any type of measurand: pressure, gas density, position, temperature, force, etc. The sensor and microcontroller can operate from the same supply voltage, so the sensor can in most cases connect directly to a port pin on the microcontroller.

The nominal reference frequency of the output signal from these devices is in the range of 20 Hz - 500 kHz, depending on the design. A change in the measurand away from the reference condition causes the frequency to shift by an amount that is related to the change in the measurand value. Transducers are available that have a full-scale frequency shift of 2-1. The microcontroller detects the change in frequency or period and converts it in software to the measurand value.

2. Connecting the Digital Transducer to the 8051

Normally the transducer output can be connected directly to one of the 8051 port pins. An exception would occur when the transducer signal does not restrict itself to the voltage range of -0.5 to +5.5 V.

The 8051 is not sensitive to the rise and fall times of its input signals. It detects transitions by sampling its port pins at fixed intervals (once per machine cycle), and responds to a change in the sequence of samples. If the slew rate of the transducer signal is extremely slow, noise superimposed on the signal could cause the sequence of samples to show false transitions. There could on that account be situations in which the transducer signal should be buffered through a Schmitt Trigger to square it up.

3. Timer/Counter Structure in the 8051

The 8051 has two 16-bit timer/counters: Timer 0 and Timer 1. Both can be configured in software to operate either as timers or as event counters.

In the “timer” function, the register is automatically incremented every machine cycle. Since a machine cycle in the 8051 consists of 12 clock periods, the timer is being incremented at a constant rate of 1/12 the clock frequency.

In the “counter” function, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin (T0 or T1). The way this function works is the external input pin is sampled once each machine cycle (once every 12 clock periods), and when the samples show a high in one cycle and a low in the next, the counter is incremented.

Note too that since it takes two machine cycles (24 clock periods) to recognize a 1-to-0 transition, the maximum count rate is 1/24 the clock frequency. If the clock frequency is 12 MHz, the maximum count rate is 500 kHz. There are no requirements on the duty cycle of the signal being counted.

The 8052, an enhanced version of the 8051, has three 16-bit timer/counters, two of which are identical to those in the 8051. The third timer/counter can operate either as a 16-bit timer/counter with automatic reload to a preset 16-bit value on rollover, or as a 16-bit timer/counter with a “capture” mode. In the capture mode a 1-to-0 transition at the T2EX pin causes the current value in the counting register to the “captured” into RAM. The third timer makes the 8052 particularly easy to interface with resonant transducers.

4. Whether to Measure Frequency or Period

Measuring the frequency requires counting transducer pulses for a fixed sample time. Measuring the period requires measuring elapsed time for a fixed number of transducer pulses. For a given level of accuracy in the determination of the value of the measurand, it is usually faster to measure the period, rather than the frequency, even if the measurand is...
proportional to frequency rather than period. However, both types of measurements will be discussed here.

Two timer/counters can be used, one to mark time and the other to count transducer pulses. If the frequency being counted does not exceed 50 kHz or so, one may equally well connect the transducer signal to an external interrupt pin, and count transducer pulses in software. That frees one timer, with very little cost in CPU time.

V. HOW TO MEASURE TRANSUDER FREQUENCY

Measuring the frequency means counting transducer pulses for some desired sample time. The count that is directly obtained is \( T \times F \), where \( T \) is the sample time and \( F \) is the frequency. The full scale range is \( T \times (F_{\text{max}} - F_{\text{min}}) \). For \( n \)-bit resolution

\[
1 \text{ LSB} = \frac{T \times (F_{\text{max}} - F_{\text{min}})}{2^n}.
\]

Therefore, the sample time required for \( n \)-bit resolution is

\[
T = \frac{2^n \times F_{\text{max}} - F_{\text{min}}}{F_{\text{max}} - F_{\text{min}}}.
\]

For example, 8-bit resolution in the measurement of a frequency that varies between 5 and 10 kHz would require, according to this formula, a sample time of 51.2 ms. The maximum acceptable frequency count would be 51.2 ms \( \times 10 \) kHz = 512 counts. The minimum would be 256 counts. Subtracting 256 from each frequency count would allow the frequency to be reported on a scale of 0 to FF in hex digits.

If \( F_{\text{min}} \) and \( F_{\text{max}} \) are closer together it takes more time to resolve them. 8-bit resolution in the measurement of a frequency that varies between 7 and 9 kHz would require a sample time of 128 ms. The maximum and minimum acceptable counts would be 1152 and 896. Subtracting 896 from each frequency count would allow the frequency to be reported on a scale of 0 to FF in hex digits.

To implement the measurement, one timer is used to establish the sample time. In this function it autoincrements every machine cycle. A machine cycle consists of 12 periods of the clock oscillator. The sample time can be converted to machine cycles by multiplying it by \( (F_{\text{t}})/12 \), where \( F_{\text{t}} \) is the 8051 clock frequency. The timer needs to be preset so that it rolls over at the end of each sample time. Then it generates an interrupt, and the interrupt routine reads and clears the transducer pulse counter, and then reloads the timer with the correct preset value.

The preset or reload value is the two's complement negative of the sample time in machine cycles. For example, with a 12-MHz clock frequency, the reload value required to establish a 51.2 ms sample time is

\[
- \frac{(51.2 \text{ ms}) \times (12000 \text{ kHz})}{12} = -51200 = 3800 \text{ H}.
\]

In many cases the required sample time exceeds the capacity of a 16-bit timer. For example, establishing a 128 ms sample time with a 12-MHz clock frequency requires a 3-byte timer with a reload of FE0C00H. The 8051 timer, being only 2-bytes wide, can be augmented in software in the timer interrupt routine to three bytes. The 8051 has a DJNZ instruction (decrement and jump if not zero) which makes it easier to code the third timer byte to count down instead of up. If the third timer byte counts down, its reload value is the two's complement of what it would be for an up-counter. For example, if the two's complement of the sample time is FE0C00H, then the reload value for the third timer byte would be 02, instead of FE. The timer interrupt routine might then be

```
DJNZ THIRD_TIMER_BYTE.OUT
MOV TL0,#0
MOV TH0,#0CH
MOV THIRD_TIMER_BYTE,#02
```

OUT: RETI

Interrupt latency will have no effect on the measurement if the latency is the same for every sample time.

The trouble with measuring the frequency is it is not only slow, but a waste of the resolving power of the 8051's timers. A timer with microsecond resolution is being used to mark off 100-ms time periods. The technique is nevertheless useful if the timer is already serving other purposes (servicing a display, perhaps), so that the sample time is coming relatively free of charge. But in most cases it is faster and equally accurate to measure the frequency by deriving it from a measurement of the period.

VI. HOW TO MEASURE THE PERIOD

Measuring the period of the transducer signal means measuring the total elapsed time over \( N \)-transducer pulses. The quantity that is directly measured is \( N \times T \), where \( T \) is the period of the transducer signal in machine cycles. The relationship between \( T \) in machine cycles and the transducer frequency \( F \) in arbitrary frequency units is

\[
T = \frac{F_{\text{t}}}{F} \times (1/12)
\]

where \( F_{\text{t}} \) is the 8051 clock frequency, in the same unit as \( F \).

The full scale range then is \( N \times (T_{\text{max}} - T_{\text{min}}) \). For \( n \)-bit resolution

\[
1 \text{ LSB} = \frac{N \times (T_{\text{max}} - T_{\text{min}})}{2^n}
\]

Therefore, the number of periods over which the elapsed time should be measured is

\[
N = \frac{2^n}{T_{\text{max}} - T_{\text{min}}}
\]

However, \( N \) must also be an integer. It is logical to evaluate the above formula (do not forget that \( T_{\text{max}} \) and \( T_{\text{min}} \) have to be in machine cycles) and select for \( N \) the next higher integer. This selection gives a period measurement that has somewhat more than \( n \)-bit resolution, which may or may not be acceptable, depending on the overall requirements of the
system. It can be scaled back to n-bit resolution, if necessary, by the following computation:

\[
\text{reported value} = \frac{NT - NT_{\text{min}}}{NT_{\text{max}} - NT_{\text{min}}}
\]

where \(NT\) is the elapsed time measured over \(N\) periods.

The computation can be done in math if a suitable divide routine is available in the software. For 8-bit resolution it is entirely reasonable to find the reported value in a look-up table, which would take up somewhat more than one page in ROM. In fact, the look-up table would contain \(NT_{\text{max}} - NT_{\text{min}}\) entries.

For example, suppose we want 8-bit resolution in the measurement of the period of a signal whose frequency varies from 5 to 10 kHz. If the clock frequency is 12 MHz, then \(T_{\text{max}}\) is \((12,000 \text{ kHz})/(12 \times 5 \text{ kHz}) = 200\) machine cycles, and \(T_{\text{min}}\) is 100 machine cycles. The timer needs to be on then for \(N = 2.56\) periods, according to the formula. Using \(N = 3\) periods will give maximum and minimum \(NT\) values of 600 and 300 machine cycles. This is somewhat more than 8-bit resolution. It can be scaled to 8 bits with a 300-byte look-up table, if desired.

To implement the measurement, one timer is used to measure the elapsed time \(NT\). Enabling its interrupt is optional. The timer interrupt could be used to indicate a short or open in the transducer circuit.

Then the transducer is connected to one of the external interrupt pins (INT0 or INT1), and this interrupt is configured to the transition-activated mode. In the transition-activated mode every 1-to-0 transition in the transducer output will generate an interrupt. The interrupt routine counts transducer pulses, and when it gets to the predetermined \(N\), it reads and clears the timer. For example

\[
\text{DINZ PULSES,OUT}
\]

\[
\text{MOV PULSES, N, PERIODS}
\]

(Read and clear timer.)

OUT: RETI

If other interrupts are also to be enabled, the one connected to the transducer should be set to Priority 1, and the others to Priority 0. This is to control the interrupt response time. The response time will not affect the measurement if it is the same for every measurement. Variations in the response time will, however, affect the measurement. Setting the pulse-counter interrupt to Priority 1 and all others to Priority 0 will minimize variations in the response time. The response time will then be limited to range from 3 to 8 machine cycles.

VII. PULSEWIDTH MEASUREMENTS

The 8051 timers have an operating mode which is particularly suited to pulswidth measurements, and may be useful here if the transducer has a fixed duty cycle, or if the transducer output is pulswidth modulated instead of frequency modulated by the measurand.

In this mode the timer is turned on by the on-chip circuitry in response to an input high at the external interrupt pin, and off by an input low. The external interrupt itself is enabled, so the same 1-to-0 transition from the transducer that turns off the timer also generates an interrupt. The interrupt routine would then read and reset the timer.

The advantage of this method is that the transducer signal has direct access to the timer gate, with the result that variations in the interrupt response time cease to be a factor. The timer can be read and cleared any time before the next high in the transducer output.

VIII. DERIVING FREQUENCY FROM A PERIOD MEASUREMENT

We now consider the problem of measuring the transducer frequency to \(n\)-bit resolution by deriving it from a direct measurement of the period. The advantage of this technique is speed. It is always faster to measure period than frequency. But it is important to end up with a frequency value that has the same resolution and accuracy as a directly measured frequency. Two questions need to be addressed.

1) To achieve \(n\)-bit resolution in the calculated frequency, how much resolution is required in the period?

2) Having measured the period to the required resolution, what is the most efficient way to calculate the frequency?

These questions will be addressed one at a time.

IX. RESOLUTION REQUIREMENTS

In general, \(n\)-bit resolution in the frequency derivation requires somewhat more than \(n\)-bit resolution in the period measurement. How much more? It will be demonstrated presently that if the transducer frequency varies over a 2-to-1 range, the frequency can be calculated with \(n\)-bit resolution from period measurements that have \((n + 1)\)-bit resolution.

The more practical form of the question is over how many periods \((N)\) must the transducer signal be sampled to obtain the required resolution in \(F\)? And so, we commence a calculation of \(N\).

The basic calculation of frequency from \(N \times T\) (which we shall call \(NT\)) is straightforward:

\[
F = \frac{N}{NT}
\]

The relationship between an increment \(dF\) in the calculated frequency due to an increment \(d(NT)\) in the measured period is, therefore,

\[
dF = \frac{N}{(NT)^2} \cdot d(NT)
\]

\[
= \frac{F^2}{N} \cdot d(NT).
\]

This equation says the value of an LSB in the calculated frequency is \((F^2)/N\) \times the value of an LSB in \(NT\). Then the maximum value that an LSB in the calculated frequency can have is \((F_{\text{max}})^2/N\) \times the value of an LSB in \(NT\). For the calculated frequency to have \(n\)-bit resolution over the entire range of frequencies, the value of its LSB must never exceed \((F_{\text{max}} - F_{\text{min}})/2^n\). Therefore, the measurement requires

\[
\frac{(F_{\text{max}})^2}{N} \times (1 \text{ LSB in } NT) \leq \frac{F_{\text{max}} - F_{\text{min}}}{2^n}.
\]

270434-3
The required resolution in $NT_{max}$ is, therefore,

$$1 \text{ LSB in } NT = \frac{N \times (F_{max} - F_{min})}{2^n} \times (F_{max})^2.$$ 

Now, to say that $NT$ is measured with $m$-bit resolution means

$$1 \text{ LSB in } NT = \frac{N \times (1/F_{min} - 1/F_{max})}{2^n}.$$ 

Substituting this value for 1 LSB into the required resolution and solving for $2^n$ yields

$$2^n \geq \frac{F_{max}}{F_{min}}.$$ 

Then the requirement on $m$ is

$$m \geq n + \frac{\ln (F_{max}/F_{min})}{\ln (2)}.$$ 

It can be stated with some certainty, then, that if the transducer frequency varies over a range of 2-to-1, the frequency can be calculated with 8-bit resolution from a period measurement that has 9-bit resolution. If the frequency variation is less than 2-to-1, another full bit of resolution in the period measurement is not needed.

To obtain $m$-bit resolution in $NT$, $N$ must satisfy

$$N \leq \frac{2^n}{T_{max} - T_{min}}.$$ 

Substituting for $2^n$, and using $T_{max} = 1/F_{min}$ and $T_{min} = 1/F_{max}$, gives the result that

$$N \leq \frac{(F_{max})^2}{F_{max} - F_{min}}.$$ 

It should be noted that the units of frequency here are periods/machine cycle, since the 8051 measures time by counting machine cycles. The conversion factor between Hz and periods/machine cycle is 12/clock frequency). So the requirement on $N$ can also be written

$$N \leq \frac{F_{max}}{F_{max} - F_{min}} \times \frac{F_{max}}{F_{xtal}} \times 12 \times 2^n.$$ 

where $F_{xtal}$ is the 8051 clock frequency in the same units as $F_{max}$ and $F_{min}$. This is the number of transducer pulses over which the transducer signal must be sampled to achieve the required solution in $F$.

For example, suppose that 8-bit resolution is required in $F$, where $F_{max} = 10 \text{ kHz}$ and $F_{min} = 5 \text{ kHz}$, and that $F_{xtal} = 12 \text{ MHz}$. Then the above calculation shows that $N = 6$ periods gives sufficient resolution in the period measurement to satisfy the resolution requirement in $F$. Six periods will take 0.6-1.2 ms of sampling time, on that frequency range. Recall that the sample time for a direct frequency measurement of the same signal and to the same resolution was earlier calculated to be 51.2 ms.

X. Computing the Frequency from the Period
The period measurement leaves one with a 16-bit integer, which is $N \times T_{max} + N$ in machine cycles. The conversion to frequency is straightforward:

$$F = \frac{N/(NT)}{NT} \text{ periods/machine cycle.}$$

The quantity of interest is probably not $F$, but a normalized measure of the amount by which $F$ exceeds its minimum acceptable value. This quantity represents, through the transducer’s transfer function, the “reported value” of the measurand, and this quantity is an $n$-bit integer whose value ranges from 0 (all bits = 0) to full scale (all bits = 1). This normalized frequency is

$$f = \frac{F - F_{min}}{F_{max} - F_{min}}.$$ 

Using $F = N/(NT)$ and $F_{min} = F_{max}/(N_{max} - NT)$ allows the normalized frequency to be written

$$f = \frac{F_{min} \times NT_{max} - NT}{F_{max} - F_{min} \times NT}.$$ 

To get a handle on what kinds of numbers are involved here, consider the situation where 8-bit resolution is required in $f$, and in which $F_{xtal} = 12 \text{ MHz}$, $F_{max} = 10 \text{ kHz}$, and $F_{min} = 5 \text{ kHz}$. We have previously determined that for this set of conditions, $N = 6$ periods gives sufficient resolution in the period measurement to satisfy the resolution requirement in $F$ (and in $f$). With a 12 MHz clock frequency, $T_{max}$ in machine cycles is $(12,000 \text{ kHz})/(12 \times 5 \text{ kHz}) = 200$ machine cycles, so $NT_{max}$ is $6 \times 200 = 1200$ machine cycles. The calculation for $f$ then becomes

$$f = \frac{1200 - NT}{NT}.$$ 

The minimum acceptable value that $NT$ can have is $(N \times T_{min}) + 1$, where $T_{min} = (12,000 \text{ kHz})/(12 \times 10 \text{ kHz}) = 100$ machine cycles. Then $N \times T_{min} = 6 \times 100 = 600$ machine cycles. The allowable values for $NT$ are then 601-1200 machine cycles, a total of 599 different values.

The fastest way to “calculate” $f$ would be with a 599-byte look-up table. This method has the added advantage that nonlinearities in the transfer function between frequency and measurand can be built into the table. Look-up tables are facilitated in the 8051 by the MOVCA, @A+PC, and MOVCA, @A+DPTR instructions. DPTR is a 16-bit “data pointer” register in the 8051. Its two bytes can be individually addressed as DPL (low byte) and DPH (high byte).

In the example under discussion, it will be necessary to load DPTR with the address of the first byte of the look-up table, less 601, plus the 2-byte value of $NT$. The software that accesses the table might then take the following form:

**TABLE EQU** (address of first table entry)
DIVIDE ROUTINE

The divide routine simulates

\[
\text{Quotient} = \frac{\text{Numerator}}{\text{Denominator}}
\]

where numerator and denominator are integers and numerator/denominator quotient is of the form

\[
\text{Quotient} = q_0 q_1 q_2 q_3 q_n
\]

Fig. 1. A divide algorithm.

MOV A,#LOW(TABLE-601)
ADD A,N7LO
MOV DPL,A
MOV A,#HIGH(TABLE-601)
ADDC A,N7HI
MOV DPH,A
CLR A
MOVC A, @A + DPTR.

At this point the accumulator contains the 8-bit value of \( f \).

It is perfectly reasonable to decide that a 599-byte look-up table is unwieldy. Its advantages are speed and built-in error correction. But a reasonably fast divide algorithm can be written to this specific purpose, making use of a priori knowledge about the sizes of the numbers that are involved in the computation. It helps to know that in this example the numerator is never going to be larger than 599 and the denominator is always greater than the numerator.

A complete discussion of divide routines is beyond the scope of this paper, but a suitable divide algorithm for this specific application is shown in Fig. 1. Reference [1] calls this the Restoring division algorithm. Its advantage is that it gates the timer off when the interrupt is generated, rather than when the interrupt is responded to.

XI. ACCURACY AND RESOLUTION

The accuracy with which the 8051 will measure the frequency or period of the transducer signal depends on two things: the accuracy of the clock oscillator and variations in the interrupt response time.

Since the clock signal is normally generated by a crystal oscillator, the oscillator accuracy normally far exceeds the quantizing error inherent in the finite \((n\text{-bit})\) resolution.

As was previously mentioned, interrupt response time does not introduce an error into the measurement itself, but variations in the interrupt response time can. Interrupt response time in the 8051 can vary from 3 to 8 machine cycles, depending on what instruction is in progress at the time the interrupt is generated. This would represent an error of \( \pm 5 \) counts in the measured value of \( NT \) during a period measurement. An error of \( \pm 5 \) counts in \( NT \) does not necessarily translate to \( \pm 5 \) LSB's in the final result, but it might still represent an error that exceeds the resolution.

In a direct frequency measurement variations in the interrupt response time would represent an error of \( \pm 5 \) in the sample time. If these kinds of errors are unacceptable there are ways to deal with them. In period measurements, if the duty cycle of the transducer is constant, the pulsewidth measurement technique, previously described, can be used. Its advantage is that it gates the timer off when the interrupt is generated, rather than when the interrupt is responded to.

In other cases one can simply increase the sample time above the minimum required to obtain the desired resolution. For example, if the measurement requires 8-bit resolution, one can design the software for an 11-bit resolution and truncate the result to 8 bits.

REFERENCES

Microcontrollers are rapidly becoming the backbone of silicon computing systems. From a technical standpoint, the most significant attribute, aside from the inclusion of RAM and ROM, that segregates a microcontroller from a microprocessor is I/O manipulation. In general, I/O manipulation is an intimate part of a microcontroller's architecture. The instruction set and architecture of a microcontroller allows the CPU to directly control the I/O facilities on the device. This is in direct contrast to a microprocessor where the I/O is essentially a "sea" of addresses and it is up to the hardware designer to place some type of I/O hardware in this I/O "sea". It should be obvious that simply adding ROM and RAM to a microprocessor WILL NOT create a microcontroller.

This intimate contact with I/O gives the microcontroller a distinct advantage over the microprocessor in applications that are I/O intensive. Microcontrollers can test, set, complement, or clear I/O port pins much faster than a microprocessor and they can also make decisions, based on the state of other hardware features, such as timer/counters with equal speed. This integration of I/O, in both hardware and software makes the microcontroller "ideal" for many types of intelligent instrumentation.

Intel's MCS-51 series of microcontrollers contain many features that can be integrated directly into many types of instruments. TABLE 1 is a brief listing of these features. To illustrate the power of the 8051 this paper will elaborate on two techniques for performing analog to digital (A to D) conversion. Both of these examples assume that some additional hardware is attached to the I/O pins of the 8051.

S/A CONVERSION TECHNIQUES

Successive approximation analog to digital conversion involves a "binary search" of an unknown voltage relative to a "fixed" known reference. The reference is selectively divided by multiples of two until the desired accuracy is reached. Figure 1 is a flowchart of a successive approximation converter. This technique usually requires a digital to analog converter to divide the reference voltage and a voltage comparator to compare the unknown voltage to the "divided" reference. Digital to analog converters and voltage comparators are readily available and relatively inexpensive. A block diagram of an 8051 based A to D converter is shown in Figure 2.

Many industrial A to D converters require 12 bits of accuracy. A 12 bit converter provides good "dynamic range" and is capable of resolving 1 part in 4096. If the applied input voltage ranges from 0 to 10 Volts, a 12 bit converter can resolve 2.4 millivolts within this range. The theoretical accuracy of a 12 bit converter is .024% +/- 1/2 least significant bit.

The power of the 8051 in this type of application is best revealed by examining the software required to implement the successive approximation algorithm. The routine for the 8051 is shown in Table 2.

The execution times given assume a 12 MHz crystal. Compare this to the following routine which is a 4 MHz 2-80
TABLE 2. SUCCESSIVE APPROXIMATION ROUTINE FOR THE 8051.

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>BYTES</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>;CLEAR PORT PINS ;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV P1,#0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ANL P2,#0FH</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>;START CONVERSION ;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P2.3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P2.3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P2.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P2.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P2.1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P2.1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L3:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P2.0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P2.0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L4:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L5:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L6:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L7:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L8</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L8:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L9</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L9:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L10:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L11:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETB P1.0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JNB P2.4,L2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CLR P1.0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L12:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONVERSION COMPLETE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TOTAL 90 46 US

NOTE: TIMING IS TYPICAL
WORST CASE = 52 US
BEST CASE = 40 US
executing the same algorithm with the D to A hardware attached to an I/O port is shown in Table 3 (assume that all bits on PORT3 are grounded, except the comparator input).

**TABLE 3. SUCCESSIVE APPROXIMATION ROUTINE FOR THE Z-80.**

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>BYTES</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>;CLEAR PORT PINS ;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD A,0</td>
<td>2</td>
<td>1.75</td>
</tr>
<tr>
<td>OUT (PORT1),A</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>OUT (PORT2),A</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>;START CONVERSION ;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD A,08H</td>
<td>2</td>
<td>1.75</td>
</tr>
<tr>
<td>OUT (PORT2),A</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>IN A,(PORT3)</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>OR A</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>IN A,(PORT2)</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>JP Z,L1</td>
<td>3</td>
<td>2.50</td>
</tr>
<tr>
<td>AND OF7H</td>
<td>2</td>
<td>1.75</td>
</tr>
<tr>
<td>L1: OR 04H</td>
<td>2</td>
<td>1.75</td>
</tr>
<tr>
<td>OUT (PORT2),A</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>IN A,(PORT3)</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>OR A</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>IN A,(PORT2)</td>
<td>2</td>
<td>2.75</td>
</tr>
<tr>
<td>JP Z,L2</td>
<td>3</td>
<td>2.50</td>
</tr>
<tr>
<td>AND OFBH</td>
<td>2</td>
<td>1.75</td>
</tr>
<tr>
<td>L2: OR 02H</td>
<td>2</td>
<td>1.75</td>
</tr>
</tbody>
</table>

**TOTAL** | 179 | 180 US |

AGAIN TIMING IS TYPICAL

WORST CAST = 190.25 US

BEST CASE = 169.25 US

One may argue that by "memory mapping" the Z-80's I/O ports the execution time could be enhanced because the user could take advantage of the Z-80's SET and RESET memory BIT instructions. In reality, a few bytes of memory are saved, but very little time! This is because the Z-80's memory oriented BIT instructions are VERY slow, requiring between 3 and 5 microseconds with a 4 MHz clock!

This is not to say that the Z-80 isn't a credible 8-bit processor. The weakness is that decisions (i.e. JUMPS) cannot be made directly on the state of a given I/O pin. JUMP instructions, on most processors, are made on the state of the flags - after some type of logical or arithmetic operation! This means that information must be moved to an internal CPU register before a decision can be made. This "moving" of information back and forth between internal registers and I/O makes the microprocessor quite inefficient, relative to the microcontroller when I/O manipulation is involved. Note that with the 8051 algorithm never "moves" data from one location to another - it directly sets, tests, and clears bits. This characteristic gives the 8051 its distinct execution advantage.

Another strength of the 8051 in this type of application, relates to the fact that I/O port pins can be set, cleared, complemented, and tested with the same speed that a microprocessor can act on it's internal registers. Note that the 8051 takes only 1 microsecond to fetch an opcode and set or-clear a port pin. A microprocessor must first fetch and decode the opcode, then place the appropriate I/O or memory address on the bus, then perform the necessary operation. All of this "communication" over the microprocessor bus significantly slows down the microprocessor.

**DUAL SLOPE INTEGRATING CONVERTER**

Integrating A to D converters operate by an indirect method of converting a voltage to a time period, then measuring the time period with a counter. Integrating techniques are quite slow, relative to successive approximation, but they are capable of providing very accurate measurements - 5 1/2 or more decimal digits - if proper analog techniques are employed. They also have the added advantage of allowing the integration period to be a multiple of 60 Hz (16.67 ms) which can eliminate inaccuracies caused by the ever present "power line". Virtually all digital voltmeters use some type of integrating technique. Figure 3 is a block diagram of a typical integrating...
A to D converter.

Figures 4A, 4B, and 4C show the three typical phases involved in the dual slope technique. Figure 4A illustrates the auto-zero phase. In this phase the integrating “loop” is closed and the offset of the analog integrator is accumulated in C auto zero. In Figure 4B, the input switch is closed and the integrator integrates the input voltage for a fixed time period T1. In figure 4C, the reference switch is closed and the integrator integrates the reference voltage until the comparator senses a zero crossing condition. The time it takes for this phase to occur is directly proportional to the amplitude of the input voltage. Additional circuitry can be added to determine the polarity of the input voltage, then switch in a reference of opposite polarity, but the basic technique remains the same.

The 8051 is an ideal controller for an intelligent integrating A to D system. The 16 bit timer/counters can provide better than 4 1/2 decimal digits of accuracy, the serial port can be used to transmit the analog reading to a printer or another processor, the CPU can be interrupted by the 60 Hz line so conversions can start at precise intervals, and software can be used to calculate and save average, peak, or RMS readings.

Another “nice” benefit of this type of converter is that very few I/O port pins are required to control the A to D hardware, so opto-isolators can be used to completely isolate the 8051.
"digital system" from the analog hardware. Opto-isolators provide an additional "bonus" in that they may provide logical level shifting if needed by the analog circuitry. Figure 5 shows how an 8051 might be connected to the analog sub-system. In practice, the analog switches can be almost anything ranging from CMOS to VFETs.

The code needed to generate the "basic" integrating A to D function is shown in Table 4.

Timer interrupts could be used so that the CPU could be doing other things while the conversion was in process. Note that very little CPU time is needed to perform the actual A to D function.

CONCLUSION

This paper illustrated possible methods of using the 8051 in A to D "instrumentation" types of applications. The power of the 8051's microcontroller architecture relates to the fact that logical "decisions" can be made directly on the state of the resident I/O hardware. This fact alone gives the 8051 a distinct advantage in "bit intensive" applications. Software and hardware support tools include in-circuit emulators, an assembler, and a high level language, PLM-51.

Presently, the 8051 is available in 3 technology "flavors"—HMOS II, HMOS-EPROM, and CHMOS, so depending on your individual application, you can have it your way.

### Table 4. Software for Integrating A to D Converter

<table>
<thead>
<tr>
<th>; ;START PROGRAM ;</th>
<th>;</th>
<th>;</th>
<th>;</th>
<th>;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR TR0</td>
<td>;;TURN TIMER OFF ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>MOV TH0,$HIGH TAZ</td>
<td>;;LOAD AUTO ZERO ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>MOV TL0,$LOW TAZ</td>
<td>;;TIME ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>ANL Pl,0F0H</td>
<td>;;MAKE A/D INACTIVE ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>SETB Pl.2</td>
<td>;;AUTO ZERO PHASE ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>SETB TR0</td>
<td>;;TURN TIMER ON ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>JNB TF0,$</td>
<td>;;LOOP TIL OVERFLOW ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>CLR TR0</td>
<td>;;TURN TIMER OFF ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>CLR TF0</td>
<td>;;RESET TOV FLAG ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>MOV TH0,$HIGH INTT</td>
<td>;;LOAD INTEGRATION ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>MOV TL0,$LOW INTT</td>
<td>;;TIME ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>CLR Pl.2</td>
<td>;;END AUTO ZERO ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>SETB Pl.1</td>
<td>;;START INTEGRATION ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>SETB TR0</td>
<td>;;START TIMER ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>JNB TF0,$</td>
<td>;;WAIT FOR OVERFLOW ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>CLR Pl.1</td>
<td>;;END INTEGRATION ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>; ;NOW, INTEGRATE THE REFERENCE ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>SETB Pl.0</td>
<td>;;AT THIS POINT TIMER 0 HAS A VALUE OF ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>; ;TWO, THE TIMER IS EQUAL TO ZERO, WHEN ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>; ;IT OVERFLOWS AND IT WAS INCREMENTED ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>; ;TWICE DURING THE LAST TWO INSTRUCTIONS ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>; ;NOW, WAIT FOR ZERO CROSS ;</td>
<td>; ; ;</td>
<td>;</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>JNB Pl.3,$</td>
<td>;;TURN THE TIMER OFF ;</td>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>CLR TR0</td>
<td>;;</td>
<td>; ; ;</td>
<td>; ;</td>
<td>; ;</td>
</tr>
<tr>
<td>; ;NOW, TIMER 0 - Vin + 3 COUNTS ;</td>
<td>; ; ;</td>
<td>; ;</td>
<td>; ;</td>
<td></td>
</tr>
</tbody>
</table>
Using Intel's ASIC Core Cell to Expand the Capabilities of an 80C51-Based System

MATT TOWNSEND
CPO TECHNICAL MARKETING MANAGER
INTRODUCTION

Intel's new ASIC family of microcontroller core cells extends the capability of the MCS®-51 product, and allows the ASIC designer more flexibility than the popular microcontroller product. This note will discuss many of the new design possibilities inherent to the 80C51 cell-based controller. This family of cells is available with a variety of RAM and ROM configurations.

<table>
<thead>
<tr>
<th>Cell Name</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC5100</td>
<td>No ROM</td>
<td>128 Bytes RAM</td>
</tr>
<tr>
<td>UC5104</td>
<td>4K ROM</td>
<td>128 Bytes RAM</td>
</tr>
<tr>
<td>UC5108</td>
<td>8K ROM</td>
<td>128 Bytes RAM</td>
</tr>
<tr>
<td>UC5116</td>
<td>16K ROM</td>
<td>256 Bytes RAM</td>
</tr>
<tr>
<td>UC5200</td>
<td>No ROM</td>
<td>256 Bytes RAM</td>
</tr>
<tr>
<td>UC5204</td>
<td>4K ROM</td>
<td>256 Bytes RAM</td>
</tr>
<tr>
<td>UC5208</td>
<td>8K ROM</td>
<td>256 Bytes RAM</td>
</tr>
<tr>
<td>UC5216</td>
<td>16K ROM</td>
<td>256 Bytes RAM</td>
</tr>
</tbody>
</table>

Other documentation will address Intel's ASIC design environment (see reference section).

The 80C51-based ASIC cell is part of a family of cell-based functions based on popular Intel standard products. Members of the 82Cxx microprocessor support peripheral family (SP8254, SP8237, SP8259, SP8284, SP82284, SP8288 and SP82288) are also available as library elements. The standard product ASIC cores are supported by a library of over 150 logic cells, representing a broad range of SSI, MSI, and I/O functions. Another class of cell library elements is designated Special Functions. These cells are predefined complex functions such as RAM, Serial I/O, A/D Converter, and a Voltage Comparator. The Special Function and general logic element cells can also be used without a standard product core in the ASIC design. Any of the available 80C51-based cores can be integrated with logic complexities up to 5000 gates.

80C51-BASED ASIC CORE

Although the 80C51-based core is functionally identical to the standard 80C51BH microcontroller, its use as a cell in the ASIC library allows more flexibility in system design and partitioning.

Figure 1 depicts the difference between the standard pinout of the MCS-51 family and the ASIC core. In order to understand the enhancements (in an applications sense) made to the core it is useful to compare its connections to the pinout of the standard product.

The illustrations shown in this note are independent of the workstation platform used to implement the design.

The illustrations shown in this note are independent of the workstation platform used to implement the design.

The standard cell version of the 80C51 provides the designer with 116 signals for connection to application specific logic. These signals represent the full function set of the MCS-51 architecture and virtually eliminate any design trade-offs required to implement an application. Notice from Figure 1 that all of the I/O ports are separated from the other functions. In the design example, the I/O are separated into their respective inputs and outputs, leaving 32 inputs and 32 outputs for port connections into the application's logic. The most immediate impact of demultiplexing the I/O of the device is that much of the logic required to complete an application is eliminated. For example, when separating the address from the data on the AD-bus, an octal latch is required. For an 80C51-based core application, the designer uses the AO–7 bus directly, thus saving approximately 100 gates. The fact that the 80C51-based core has so many connections available does not mean an application will be forced into higher pin count packages. A 80C51-based ASIC can implement many system functions more economically than a discrete implementation. The design example illustrates a system with over 280 interconnects that can be integrated into one ASIC device. This application note will illustrate the less obvious ways in which the core can be used.

The illustrations shown in this note are independent of the workstation platform used to implement the design.

Intel provides the complete test vectors necessary to test the 80C51-based ASIC core, which have been derived from the standard product 80C51 test vector set.
**RECONSTRUCTION OF STANDARD PRODUCT I/O**

When designing the 80C51-based ASIC core, Intel removed the pin multiplexers and I/O functions of the 80C51, and restructured them as companion cells. Companion cells allow the ASIC designer to reconfigure the ASIC cell to function exactly like the standard product. Alternatively, the designer can choose to reconstruct a subset of the standard product I/O or select no reconstruction at all. Consult the references for more information about the use and function of Intel’s companion cells.
MULTIPLE SOURCES OF RESET

Key to the 80C51’s core-isolation test method is the ability to put the core into a condition that can verify the processor without the user’s logic affecting the test. ERST is vital to controlling the ability to put the core based ASIC into test mode. It must be brought directly from the core to a package pin. Interface is via the PRESET companion cell. Because a dedicated reset pin may be restrictive in many applications, a second reset connection, RESET, has been included.

Including this second reset connection allows the designer to simplify the overall ASIC design. Many applications require two sources of reset, usually a power-on-clear with a watchdog timer. Previously, the designer was faced with “ORing” an RC time constant circuit with the timer logic, resulting in an implementation which was not straightforward or cost effective. Figure 2 shows an 80C51-based ASIC implementation.

A system reset, in many designs, employs an active low logic level. Since the 80C51’s reset requires an active high level, there is usually an inverter in the path to the microcontroller. It was mentioned earlier in this section that ERST must be brought directly from the core to a package pin. This is not entirely true; the inclusion of the inverter is allowed.

I/O EXPANSION WITH THE 80C51-BASED CORE

For the standard MCS-51 product, the need for I/O expansion is often due to the need for external memory and/or port expansion. The designer’s use of the on-chip peripherals (eg. Serial I/O or Interrupts) often leaves only Port 1 intact.

Figure 2. Multiple Sources of Reset for 80C51-Based ASIC Core
Figure 3. Simple MCS®-51 I/O Expansion with 8255

Figure 3 depicts one case where the 80C51 can gain an expanded set of I/O ports. In addition to requiring additional package pins, this implementation would require more power supply capacity and passive components (bypass capacitors) than would be necessary if the I/O expansion were to be included on-chip with the microcontroller. Not only is PC board size decreased, but the overall system reliability increases with the ASIC solution. In addition, the 8255 port expander, being a highly flexible device, requires software to configure the device to the application.

The simplest way to add I/O ports with the core is by way of a direct connection from the core's IP or OP signals through I/O functions selected from the cell library and connected to package pins. See Figure 4. In this example, decoding is very simple and the component count is minimal.
This technique represents the most silicon and program code efficient way to implement I/O with the core. Program code is composed of MOV operations rather than the MOVX needed for any Memory mapped I/O implementations. Logical operations to the port (ANL, ORL, XRL) can still be used. It is not necessary to update or maintain indirect pointers. Since quasi-bidirectional cells are not used, it is not necessary to set a “1” to the port in order to set these cells. Eliminating MOVX and port setup operations could result in significant codespace savings.

The drawback to the direct approach is that program code written for the 80C51 using memory mapped I/O is not directly transportable to the core design. In most cases, however, implementing I/O expansion that allows code transportability is a simple task with a 80C51-based ASIC.

Most support peripherals are designed to be configurable for many different operating conditions. This is certainly true for a device like the 8255 as well; the port signals can be programmed as inputs, outputs, or bidirectional. In most applications the peripheral’s setup is never changed after initialization. Port pins are set to either input, output or I/O. The peripheral’s configuration is most often “set up” with data fields sent to a configuration or command register. This register is located at one of the peripheral’s selectable addresses. For a cell-based implementation where code transportability is required, recreating an 8255-like function is straightforward. Since all setup information is written to one register and setup is not required because the port signal directions are fixed, that one register can become a “bit bucket”. Figure 5 shows an example with one 8-bit input port, and one 8-bit output port, both memory-mapped. Note that while the 8255 contains three 8-bit ports, an ASIC can be implemented with the exact amount of functionality desired.

Implementing the full function set of the 8255 would result in an increased gate count (550 gates) included on the 80C51-based ASIC. While 550 gates can easily be included on the same silicon chip, implementing the “exact functionality” version using elements from the cell library would consume only 100 gates.

**ON-CHIP CLOCK GENERATION**

In many designs, the built-in crystal oscillator of the 80C51 is not utilized because the clock signal must be used for other system functions. However, the clock to the 80C51 still must be generated and driven into the X1/X2 pins. A clock generator is required somewhere in the system and is also used to clock many of the system functions surrounding the microcontroller.

For 80C51-based ASIC designs, all clocking functions, including the clock generator, can be brought on-chip. The advantages to doing so include enhanced reliability and a less costly, more noise free design. Clock generation is accomplished by the companion cell POSC (or POSC2 for frequencies between 16 MHz and 38 MHz) which can be used to drive the TICLK input connection to the core. The POSC output can be sent to user defined logic configured to generate other necessary

---

**Figure 5. I/O Mapped Like Figure 3’s 8255**
clocks in a system. Where the POSC cell fan out is high and might cause concerns about clock edge skew, a cell like BUF2 can be used. For systems where it is required that the signal be brought off-chip (formerly off-board), the on-chip generated clocks can be sent to output cells and on to package pins. Figure 6 depicts generation flexibility and clock source for a 80C51-based ASIC design.

Figure 6 illustrates a design which requires a high frequency clock to operate on a section of user-defined logic. For this, cell POSC2 is selected for the ASIC design and is set to 24 MHz. In order to meet clock specifications for the core, this 24 MHz master clock is divided down in order to provide the required 12 MHz. As discussed in the 80C51-based ASIC data sheet, the ATE must be able to drive the core’s clock directly. For test modes, the ATE-generated clock is driven to the core's TICLK connection.

Note that signal P2 is shown being used for the application’s clocks. It is sent to other logic in the ASIC and to a package pin as well.

**SHARING THE TEST BUS**

In order for Automatic Test Equipment (ATE) to exercise the 80C51-based ASIC, the bus EADB must be brought directly to package pins. A specially designed I/O cell, PADB, must be directly connected to the EADB bus to ensure testability of the core as well as the user’s logic.

Requiring the EADB bus to appear as package pins does not impose any design restrictions on 80C51-based ASICs designed to access external memory or peripherals. If your design does not call for the EADB to access external memory peripherals, the EADB may be multiplexed with user I/O. Contact your Intel Technology Center for the best implementation for your application.

Intel supplies all test programs required to completely test the ASIC core cell. Designers are required to supply test vectors that exercise their unique logic only.
OBSERVING THE CONTENTS OF THE PROGRAM COUNTER

The 80C51-based ASIC core connections A0–A15 always display the contents of the program counter (except in the case of MOVX instructions.) This feature allows another level of real time control by monitoring instruction events within the core. By attaching comparator circuitry to the program counter contents, signals can be generated to depict events within the program. Figure 7 shows such a circuit.

The discussions in this note are not intended to be an exhaustive summary of the range of design possibilities available to the ASIC designer. Rather, it is hoped that it encourages the thought process toward even more innovative uses.

The following is an example of an actual system problem and how it was resolved using a 80C51-based ASIC. The example utilizes many of the techniques discussed above.

DESIGN EXAMPLE

Figure 8 shows a typical MCS-51-based design, which includes a port expander, timer/counter chip, a high speed event counter and a low-cost EPROM containing stable code. In this application, the 8031 controls a system based on numerous timed events. Many high speed clocks are involved, making for a potentially noisy environment, and a watchdog timer has been included to provide for soft recoveries if the microprocessor program flow is upset. The watchdog circuitry is shown as a high level block.

The design must take an accurate sample of events designated at the EVENT input. The 16-bit count is read and processed under a timed interrupt designated by and generated from one of the 8254 Timers. The counter chain must be clocked at 24 MHz in order for unique and accurate event samples to occur.

Another 8254 counter is programmed for single-shot mode to provide for a strobe window for some circuitry external to the PCB assembly. An 8-bit parallel data input is required. The system must control peripherals external to this main assembly, resulting in a requirement for address decoder selection.

Note that adequate bypass capacitors are required due to the clock speeds and the high number of pin connections. (There are 272 pins, not including the WDT and Oscillator Blocks.) A multilayer PCB is also required to compensate for the amount of wires needed to connect all the components.

Figure 9 depicts the 80C51-based ASIC solution for the design in Figure 8. Note that all of the circuitry in Figure 8 is included on a single ASIC chip. Rather than use memory-mapped I/O, the design has been converted to use the core's direct ports. Note that the 8255 function has been removed and instead the UC5116 port connections are used. Some minor software changes are required, and signals required to access off-chip program memory have been provided. This figure does not show all test pin requirements; however, no additional package pins will be required. For this example, the designer could begin production runs with an EPROM. Once the application code stabilized, it could be developed and submitted to Intel for incorporation into the core. In this example, most of the high speed signals are contained within the ASIC, making the watchdog timer unnecessary. If needed, the overall cost of including it on the ASIC (= 500 gates) makes the functions relatively inexpensive to keep.

Overall system pin requirements have decreased as well. This 80C51-based ASIC can be produced using a 68-pin PLCC which may reduce the bypass capacitor requirements as well as the need for a multilayer PCB.
Comparing the two solutions:

<table>
<thead>
<tr>
<th>Component Count</th>
<th>Discrete</th>
<th>80C51-ASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum PCB Layers</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>System Reliability</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Power Supply Current</td>
<td>~3A</td>
<td>~30 mA</td>
</tr>
</tbody>
</table>

**REFERENCE DOCUMENTATION**

<table>
<thead>
<tr>
<th>Order Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>231816</td>
<td>Introduction to Cell-Based Design</td>
</tr>
<tr>
<td>83002</td>
<td>Cell-Based Design—Daisy Environment</td>
</tr>
<tr>
<td>830000</td>
<td>Cell-Based Design—Mentor Environment</td>
</tr>
<tr>
<td>210918</td>
<td>Embedded Controller Handbook</td>
</tr>
<tr>
<td>270535</td>
<td>Embedded Control Applications</td>
</tr>
</tbody>
</table>

**Figure 9. 80C51-Based ASIC Solutions**
A Fast-Turnaround, Easily Testable ASIC Chip for Serial Bus Control

Don Ellis and Shailesh Trivedi
Intel Corp
Chandler, AZ

ABSTRACT

This paper describes the standard cell ASIC design methodology for a serial bus controller chip. This is a prototype CMOS chip which was designed in 19 weeks for an automotive application. The chip includes testability circuits which help attain 98% fault coverage.

INTRODUCTION

Fast-turnaround chip design has become important in the application-specific integrated circuit (ASIC) marketplace, where low production volumes preclude long design cycles. To address this market, the ASIC design methodology relies upon automatic layout software to generate fast chip layouts, at the expense of larger die sizes and somewhat lower performance. Pre-designed standard circuit cells eliminate the need for extensive circuit simulation, further shortening the design cycle. These design techniques can produce fast prototype chips for system demonstration and debug, or production parts for low-volume applications.

Intel's Automotive Operation in Chandler, Arizona recently employed standard cell ASIC technology to produce a prototype serial bus controller chip for an automotive customer. In this paper we will describe the design methodology used to meet the 19-week design schedule for this chip, along with the testability strategy which was implemented in order to achieve a 98% fault grade.

CHIP OVERVIEW AND CONSTRUCTION

The serial bus controller is a standard cell CMOS chip that interfaces a microprocessor to a serial communication bus in an automobile. The chip performs both transmit and receive functions. The transmit function consists of a first-in, first-out (FIFO) data buffer feeding a parallel-in, serial-out (PISO) shift register, and the receive function consists of a serial-in, parallel-out (SIPO) shift register driving one port of a dual-port random access memory (DPRAM). The block diagram is shown in Figure 1.

The transmit function requires a decidedly non-standard 64 x 18 bit FIFO buffer. This is constructed with a 64 x 18 bit RAM and two address counters, as shown in Figure 2. The standard cell library did not contain a 64 x 18 bit RAM cell, so we had to construct it using an existing 64 x 8 RAM cell. We modified this cell, adding two more bits to create a 64 x 10 RAM cell, then connected it in parallel with the original 64 x 8 RAM, thus extending the word length to 18 bits. Before the 64 x 10 RAM cell could be added to the standard cell library, we had to fully characterize it using circuit simulation, like every other cell in the library. Two additional RC delay cells were also created to generate RAM read and write timings in the absence of microprocessor control signals.

The receive function requires a 1K x 8 bit dual-port RAM, but the standard cell library contained only single-port RAM cells. Fortunately, no cell modifications were necessary in this case. We used the existing 1K x 8 RAM cell, multiplexing its data and address buses to simulate dual-port operation, as shown in Figure 3. RAM read and write timings are once again generated using the RC delay cells mentioned above.

The final chip was manufactured in both single-layer metal (SLM) and double-layer metal (DLM) versions on a 1.5 micron CMOS process, resulting in a 355 x 294 mil chip with 68 I/O pins. It consists of 3 RAM arrays (9.3K bits total) and about 3,000 logic gates of control logic, for a total of 76,735 transistors. Of the 8,715 transistors contributed by the control logic, 11% belong to testability circuits which were added to increase the testability of the chip (i.e., shorten test program development time and tester run time). The testability strategy will be discussed later.
FIGURE 1. SERIAL BUS CONTROLLER BLOCK DIAGRAM

FIGURE 2. FIFO (FIRST-IN, FIRST-OUT) BUFFER
The 19 week design schedule for this chip dictated the use of design automation tools. Since the chip included 3 large RAM arrays, gate arrays were impractical, so standard cells were used with automatic placement and routing software. The automatically generated layout was transferred into Intel's full custom design system for some final edits, and the usual design rule checking and verification procedures were followed prior to mask making and processing.

A standard cell design usually proceeds through the following steps:
1. Translation of the logic into standard cells.
2. Schematic capture into a computer database.
3. Extraction of a cell interconnection "netlist".
4. Logic and timing simulation.
5. Automatic layout generation.
6. Parasitic extraction and re-simulation.

The entire design procedure is outlined in Figure 4, and each step is described briefly below.

TRANSLATION INTO STANDARD CELLS
Our first task was to translate our customer's board-level schematics into a logic design consisting of subcircuits from the standard cell library. Since the customer's schematics referenced IC packages only, this involved the detailed design of the FIFO and DPRAM blocks (described above). A major part of the task was the design of the extra standard cells mentioned above, with their characterization and inclusion in the cell library.

SCHEMATIC CAPTURE, NETLIST EXTRACTION, SIMULATION
We performed schematic capture on a Daisy Personal Logician (PC-AT based) workstation, where each of the standard cells was available as a basic circuit element. We "compiled" each schematic separately to verify its integrity, then linked them together into a complete design database. Finally, we generated a "netlist", or device interconnection list, from this database. This netlist served as input to Intel's logic simulator on our VAX, which we used to verify design correctness. The logic simulator flagged several timing and glitch problems which were corrected before proceeding to layout.

AUTOMATIC LAYOUT GENERATION
We performed layout generation using the CAL-MP program from Silvar-Lisco. Working from the netlist, the program placed the three RAM arrays according to our instructions, then arranged the remaining standard cells in rows according to its own optimization algorithm. At this point prior to signal routing, we instructed the program to further iterate its optimization steps, as we manually modified several cell placements from the graphics terminal. Once all cell placements were determined, the program performed signal and power routing automatically.
The CAL-MP program accepted layout constraints in a variety of forms. In addition to the netlist information, we defined pad placements and the number of standard cell rows, and constrained a few critical signals (such as clocks) to two vertical metal buses traversing the right and left sides of the chip. Furthermore, several unconstrained signals were assigned numerical "strengths" greater than the default of 1.0, which weighted their consideration in the optimization algorithm, tending to shorten them. We ultimately generated more than 20 layouts with widely varying signal strengths, until we were satisfied that very little further improvement was possible.

PARASITIC EXTRACTION

After a layout is generated, it must be proven to work in the presence of parasitic resistances and capacitances contributed by the signal interconnects. These parasitics are extracted from the layout and added to the netlist for a post-layout simulation cycle. In principle, each iterated layout should be re-simulated, but after about 10 layout generations we could easily predict simulation performance from the raw parasitic values.

Because the first version of this chip was fabricated in single-layer metal with a great deal of polysilicon interconnection, parasitic series resistances were just as important in limiting performance as are parasitic capacitances. Unfortunately, series resistors are difficult to systematically insert into a netlist, so we had to simulate the resulting RC delays using Intel's circuit simulator. For the double-layer metal version, we could safely ignore series resistances since the metal sheet resistance is three orders of magnitude smaller than that of polysilicon.

DESIGN FOR TESTABILITY

Our test goal for this part was a 98% fault grade, and since this was a fast-turnaround project with little time for test program development, we included a variety of testability circuits on the chip. An added benefit to this approach was that the testability circuits simplified our debugging procedures. This strategy ultimately paid off, because we were able to quickly isolate and correct a RAM timing problem on the first silicon.

Since this chip has a relatively small node count, we adopted an "ad hoc" rather than "structured" testability strategy. This means that we added test circuits on a case by case basis to improve the controllability and observability of the overall chip, rather than implementing a scan path, a built-in self test, or some other more elaborate scheme. Ad hoc testability design is appropriate for small chips having relatively low transistor/pin ratios. This chip has 8,715 transistors (excluding those in the RAMs) versus 68 pins, for a transistor/pin ratio of 128. In contrast, Intel's 80386 microprocessor has 275,000 transistors versus 132 pins for a ratio of 2,083, clearly requiring structured testability techniques.

Two pins are allocated for test purposes, which are used to select among four modes: a normal operating mode, and three test modes. This test mode strategy is shown in Figure 5. The test modes are used to partition the chip into three isolated subcircuits to be tested independently. In each mode, signals with poor visibility internal to the active subcircuit are brought out to the pads, and the non-active subcircuits are turned off by disabling their clock inputs. The test program can then exercise the active circuit, with the goal of toggling each internal node for maximum fault coverage.

Eleven of the 28 chip inputs provide test inputs in the three test modes, and 16 of the 23 chip outputs serve double duty as test outputs. Although input pins can be connected to several internal test points in parallel (usually multiplexer inputs), only one signal at a time can drive an output pin. These outputs are multiplexed using three stages of 2:1 multiplexers (one for each mode), and the outputs are collected into a 16 bit "test bus" which circumnavigates the chip.
RAM TESTABILITY

RAM testability is a special case, because a RAM is inherently fully testable provided its data and address buses are accessible, along with the necessary control signals. The difficulty here is that the RAMs are embedded in functional blocks which, especially in the FIFO, tends to disguise the inherent RAM accessibility.

Inside the DPRAM, the 1K x 8 RAM is read directly by the microprocessor using the external data and address buses, so observability is no problem. Writes, however, occur from the SIPO during serial reception. It would be particularly painful to test a 1K RAM using serial writes, so a modification was necessary to improve RAM controllability. In test mode, the address multiplexer is held to the address bus by overriding the select line, and a set of eight extra multiplexers were added to the data demultiplexer to allow bidirectional data flow into and out of the RAM. Thus, the SIPO circuit is completely bypassed in test mode.

The FIFO RAM is addressed by one of two counters in the operating mode, which presents a problem unless we are willing to accept sequential test addressing, or at least a very complex address setup procedure. The solution was to override the address bus with a multiplexer fed by input pin signals. The data bus presented the same problem as the DPRAM, but in reverse: data is written by the microprocessor using the external buses, but reads are serialized in the PISO. We decided that serial output was acceptable in FIFO test mode, however, because the FIFO has only 64 locations to test (versus 1024 in the DPRAM), and the words are 18 bits long, which would require 18 extra multiplexers. Thus, for the FIFO data, we left well enough alone.

CONCLUSION

This serial bus controller chip was designed using ASIC techniques in a very short time, resulting in a quick prototype chip which our automotive customer could use to evaluate his system design in a timely manner. The inclusion of testability circuits further shortened the engineering debug time as well as the manufacturing test time. This project demonstrates that standard cell design is an attractive fast-turnaround methodology, and that a good testability strategy provides additional benefits which outweigh the extra design effort.

ACKNOWLEDGEMENT

The authors would like to thank Graham Tubbs, for guiding us through the maze of ASIC design tools, Dinesh Maheshwari and Keith Steele, who helped prepare our final layout for processing, and Mukund Patel and Magdiel Galan who helped us test and debug the final chip.
UPI-452 Accelerates iAPX 286 Bus Performance

CHRISTOPHER SCOTT
TECHNICAL MARKETING ENGINEER
INTEL CORPORATION
INTRODUCTION

The UPI-452 targets the leading problem in peripheral to host interfacing, the interface of a slow peripheral with a fast Host or “bus utilization”. The solution is data buffering to reduce the delay and overhead of transferring data between the Host microprocessor and I/O subsystem. The Intel CMOS UPI-452 solves this problem by combining a sophisticated programmable FIFO buffer and a slave interface with an MSC-51 based microcontroller.

The UPI-452 is Intel's newest Universal Peripheral Interface family member. The UPI-452 FIFO buffer enables Host—peripheral communications to be through streams or bursts of data rather than by individual bytes. In addition the FIFO provides a means of embedding commands within a stream or block of data. This enables the system designer to manage data and commands to further off-load the Host.

The UPI-452 interfaces to the iAPX 286 microprocessor as a standard Intel slave peripheral device. READ, WRITE, CS and address lines from the Host are used to access all of the Host addressable UPI-452 Special Function Registers (SFR).

The UPI-452 combines an MSC-51 microcontroller, with 256 bytes of on-chip RAM and 8K bytes of EPROM/ROM, twice that of the 80C51, a two channel DMA controller and a sophisticated 128 byte, two channel, bidirectional FIFO in a single device. The UPI-452 retains all of the 80C51 architecture, and is fully compatible with the MSC-51 instruction set.

This application note is a description of an iAPX 286 to UPI-452 slave interface. Included is a discussion of the respective timings and design considerations. This application note is meant as a supplement to the UPI-452 Advance Data Sheet. The user should consult the data sheet for additional details on the various UPI-452 functions and features.

UPI-452 iAPX 286 SYSTEM CONFIGURATION

The interface described in this application note is shown in Figure 1, iAPX 286 UPI-452 System Block Diagram. The iAPX 286 system is configured in a local bus architecture design. DMA between the Host and the UPI-452 is supported by the 82258 Advanced DMA Controller. The Host microprocessor accesses all UPI-452 externally addressable registers through address decoding (see Table 3, UPI-452 External Address Decoding). The timings and interface descriptions below are given in equation form with examples of specific calculations. The goal of this application note is a set of interface analysis equations. These equations are the tools a system designer can use to fully utilize the features of the UPI-452 to achieve maximum system performance.

HOST-UPI-452 FIFO SLAVE INTERFACE

The UPI-452 FIFO acts as a buffer between the external Host 80286 and the internal CPU. The FIFO allows the Host - peripheral interface to achieve maximum decoupling of the interface. Each of the two FIFO channels is fully user programmable. The FIFO buffer ensures that the respective CPU, Host or internal CPU, receives data in the same order as transmitted. Three slave bus interface handshake methods are supported by the UPI-452; DMA, Interrupt and Polled.

The interface between the Host 80286 and the UPI-452 is accomplished with a minimum of signals. The 8 bit data bus plus READ, WRITE, CS, and A0-2 provide access to all of the externally addressable UPI-452 registers including the two FIFO channels. Interrupt and DMA handshaking pins are tied directly to the interrupt controller and DMA controller respectively.

DMA transfers between the Host and UPI-452 are controlled by the Host processors DMA controller. In the example shown in Figure 1, the Host DMA controller is the 82258 Advanced DMA Controller. An internal DMA transfer to or from the FIFO, as well as between other internal elements, is controlled by the UPI-452 internal DMA processor. The internal DMA processor can also transfer data between Input and Output FIFO channels directly. The description that follows details the UPI-452 interface from both the Host processor's and the UPI-452's internal CPU perspective.

One of the unique features of the UPI-452 FIFO is its ability to distinguish between commands and data embedded in the same data block. Both interrupts and status flags are provided to support this operation in either direction of data transfer. These flags and interrupts are triggered by the FIFO logic independent of, and transparent to either the Host or internal CPUs. Commands embedded in the data block, or stream, are called Data Stream Commands.

Programmable FIFO channel Thresholds are another unique feature of the UPI-452. The Thresholds provide for interrupting the Host only when the Threshold number of bytes can be read or written to the FIFO buffer. This further decouples the Host UPI-452 interface by relieving the Host of polling the buffer to determine the number of bytes that can be read or written. It also reduces the chances of overrun and underrun errors which must be processed.

The UPI-452 also provides a means of bypassing the FIFO, in both directions, for an immediate interrupt of either the Host or internal CPU. These commands are called Immediate Commands. A complete description of the internal FIFO logic operation is given in the FIFO Data Structure section.
Figure 1: IAAPX 286 UPI-452 System Block Diagram
UPI-452 INITIALIZATION

The UPI-452 at power-on reset automatically performs a minimum initialization of itself. The UPI-452 notifies the Host that it is in the process of initialization by setting a Host Status SFR bit. The user UPI-452 program must release the UPI-452 from initialization for the FIFO to be accessible by the Host. This is the minimum Host to UPI-452 initialization sequence. All further initialization and configuration of the UPI-452, including the FIFO, is done by the internal CPU under user program control. No interaction or programming is required by the Host 80286 for UPI-452 initialization.

At power-on reset the UPI-452 automatically enters FIFO DMA Freeze Mode by resetting the Slave Control (SLCON) SFR FIFO DMA Freeze/Normal Mode bit to FIFO DMA Freeze Mode (FRZ = "0"). This forces the Slave Status (SSTAT) and Host Status (HSTAT) SFR FIFO DMA Freeze/Normal Mode bits to FIFO DMA Freeze Mode In Progress. FIFO DMA Freeze Mode allows the FIFO interface to be configured, by the internal CPU, while inhibiting Host access to the FIFO.

The MODE SFR is forced to zero at reset. This disables, (tri-states) the DRQIN/INTRQIN, DRQOUT/INTRQOUT and INTRQ output pins. INTRQ is inhibited from going active to reflect the fact that a Host Status SFR bit, FIFO DMA Freeze Mode, is active. If the MODE SFR INTRQ configure bit is enabled ("1"), before the Slave Control and Host Status SFR FIFO DMA Freeze/Normal Mode bit is set to Normal Mode, INTRQ will go active immediately.

The first action by the Host following reset is to read the UPI-452 Host Status SFR Freeze/Normal Mode bit to determine the status of the interface. This may be done in response to a UPI-452 INTRQ interrupt, or by polling the Host Status SFR. Reading the Host Status SFR resets the INTRQ line low.

Any of the five FIFO interface SFRs, as well as a variety of additional features, may be programmed by the internal CPU following reset. At power-on reset, the five FIFO Special Function Registers are set to their default values as listed in Table 1. All reserved location bits are set to one, all other bits are set to zero in these three SFRs. The FIFO SFRs listed in Table 1 can be programmed only while the UPI-452 is in FIFO DMA Freeze Mode. The balance of the UPI-452 SFRs default values and descriptions are listed in the UPI-452 Advance Data Sheet in the Intel Microsystems Component Handbook Volume II and Microcontroller Handbook.

The above sequence is the minimum UPI-452 internal initialization required. The last initialization instruction must always set the UPI-452 to Normal Mode. This causes the UPI-452 to exit Freeze Mode and enables Host read/write access of the FIFO. The internal CPU sets the Slave Control (SLCON) SFR FIFO DMA Freeze/Normal Mode (FRZ) bit high (= 1) to activate Normal Mode. This causes the Slave Status (SSTAT) and Host Status (HSTAT) SFR FIFO DMA Freeze Mode bits to be set to Normal Mode. Table 2, UPI-452 Initialization Event Sequence Example, shows a summary of the initialization events described above.

**Table 1. FIFO Special Function Register Default Values**

<table>
<thead>
<tr>
<th>SFR Name</th>
<th>Label</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Boundary Pointer</td>
<td>CBP</td>
<td>40H/64D</td>
</tr>
<tr>
<td>Output Channel Read Pointer</td>
<td>ORPR</td>
<td>40H/64D</td>
</tr>
<tr>
<td>Output Channel Write Pointer</td>
<td>OWPR</td>
<td>40H/64D</td>
</tr>
<tr>
<td>Input Channel Read Pointer</td>
<td>IRPR</td>
<td>00H/00D</td>
</tr>
<tr>
<td>Input Channel Write Pointer</td>
<td>IWPR</td>
<td>00H/00D</td>
</tr>
<tr>
<td>Input Threshold</td>
<td>ITH</td>
<td>00H/00D</td>
</tr>
<tr>
<td>Output Threshold</td>
<td>OTH</td>
<td>01H/1D</td>
</tr>
</tbody>
</table>

**Table 2. UPI-452 Initialization Event Sequence Example**

<table>
<thead>
<tr>
<th>Event Description</th>
<th>SFR/bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-on Reset</td>
<td></td>
</tr>
<tr>
<td>UPI-452 forces FIFO DMA Freeze Mode (Host access to FIFO inhibited)</td>
<td>SLCON FRZ = 0</td>
</tr>
<tr>
<td>UPI-452 forces Slave Status and Host Status SFR to FIFO DMA Freeze Mode In Progress</td>
<td></td>
</tr>
<tr>
<td>UPI-452 forces all SFRs, including FIFO SFRs, to default values.</td>
<td></td>
</tr>
<tr>
<td><strong>Host READ's UPI-452 Host Status (HSTAT) SFR to determine interrupt source, INTRQ goes low</strong></td>
<td>MODE MD4 = 1</td>
</tr>
<tr>
<td><strong>Host polls Host Status SFR to determine when it can access the FIFO</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Host waits for UPI-452 Request for Service interrupt to access FIFO</strong></td>
<td></td>
</tr>
<tr>
<td><strong>user option</strong></td>
<td></td>
</tr>
</tbody>
</table>
FIFO DATA STRUCTURES

Overview

The UPI-452 provides three means of communication between the Host microprocessor and the UPI-452 in either direction;

- Data
- Data Stream Commands
- Immediate

Data and Data Stream Commands (DSC) are transferred between the Host and UPI-452 through the UPI-452 FIFO buffer. The third, Immediate Commands, provides a means of bypassing the FIFO entirely. These three data types are in addition to direct access by either Host or Internal CPU of dedicated Status and Control Special Function Registers (SFR).

The FIFO appears to both the Host 80286 and the internal CPU as 8 bits wide. Internally the FIFO is logically nine bits wide. The ninth bit indicates whether the byte is a data or a Data Stream Command (DSC) byte; 0 = data, 1 = DSC. The ninth bit is set by the FIFO logic in response to the address specified when writing to the FIFO by either Host or internal CPU. The FIFO uses the ninth bit to condition the UPI-452 interrupts and status flags as a byte is made available for a Host or internal CPU read from the FIFO. Figures 2 and 3 show the structure of each FIFO channel and the logical ninth bit.

It is important to note that both data and DSCs are actually entered into the FIFO buffer (see Figures 2 and 3). External addressing of the FIFO determines the state of the internal FIFO logic ninth bit. Table 3 shows the UPI-452 External Address Decoding used by the Host and the corresponding action. The internal CPU interface to the FIFO is essentially identical to the external Host interface. Dedicated internal Special Function Registers provide the interface between the FIFO, internal CPU and the internal two channel DMA processor. FIFO read and write operations by the Host and internal CPU are interleaved by the UPI-452 so they appear to be occurring simultaneously.

The ninth bit provides a means of supporting two data types within the FIFO buffer. This feature enables the Host and UPI-452 to transfer both commands and data while maintaining the decoupled interface a FIFO buffer provides. The logical ninth bit provides both a means of embedding commands within a block of data and a means for the internal CPU, or external Host, to discriminate between data and commands. Data or DSCs may be written in any order desired. Data Stream Commands can be used to structure or dispatch the data by defining the start and end of data blocks or packets, or how the data following a DSC is to be processed.

A Data Stream Command (DSC) acts as an internal service routine vector. The DSC generates an interrupt to a service routine which reads the DSC. The DSC byte acts as an address vector to a user defined service routine. The address can be any program or data memory location with no restriction on the number of DSCs or address boundaries.

A Data Stream Command (DSC) can also be used to clear data from the FIFO or “FLUSH” the FIFO. This is done by appending a DSC to the end of a block of data entered in the FIFO which is less than the programmed threshold number of bytes. The DSC will cause an interrupt, if enabled, to the respective receiving CPU. This ensures that a less than Threshold number of bytes in the FIFO will be read. Two conditions force a Request for Service interrupt, if enabled, to the Host. The first is due to a Threshold number of bytes having been written to the FIFO Output channel; the second is if a DSC is written to the Output FIFO channel. If less than the Threshold number of bytes are written to the Output FIFO channel, the Host Status SFR flag will not be set, and a Request for Service interrupt will not be generated, if enabled. By appending a DSC to end of the data block, the FIFO Request for Service flag and/or interrupt will be generated.

An example of a FIFO Flush application is a mass storage subsystem. The UPI-452 provides the system interface to a subsystem which supports tape and disk storage. The FIFO size is dynamically changed to provide the maximum buffer size for the direction of transfer. Large data blocks are the norm in this application. The FIFO Flush provides a means of purging the FIFO of the last bytes of a transfer. This guarantees that the block, no matter what its size, will be transmitted out of the FIFO.

Immediate Commands allow more direct communication between the Host processor and the UPI-452 by bypassing the FIFO in either direction. The Immediate Command IN and OUT SFRs are two more unique address locations externally and internally addressable. Both DSCs and Immediate Commands have internal interrupts and interrupt priorities associated with their operation. The interrupts are enabled or disabled by setting corresponding bits in the Slave Control (SLCON), Interrupt Enable (IEC), Interrupt Priority (IPC) and Interrupt Enable and Priority (IEP) SFRs. A detailed description of each of these may be found in the UPI-452 Advance Information Data Sheet.
Figure 2. Input FIFO Channel Functional Diagram
Figure 3. Output FIFO Channel Functional Diagram
Below is a detailed description of each FIFO channel's operation, including the FIFO logic response to the ninth bit, as a byte moves through the channel. The description covers each of the three data types for each channel. The details below provide a picture of the various FIFO features and operation. By understanding the FIFO structure and operation the user can optimize the interface to meet the requirements of an individual design.

**OUTPUT CHANNEL**

This section covers the data path from the internal CPU to the HOST. Data Stream Command or Immediate Command processing during Host DMA Operations is covered in the DMA section.

### UPI-452 Internal Write to the FIFO

The internal CPU writes data and Data Stream Commands into the FIFO through the FIFO OUT (FOUT) and Command OUT (COUT) SFRs. When a Threshold number of bytes has been written, the Host Status SFR Output FIFO Request for Service bit is set and an interrupt, if enabled, is generated to the Host. Either the INTRQ or DRQOUT/INTRQOUT output pins can be used for this interrupt as determined by the MODE and Host Control (HCON) SFR setting. The Host responds to the Request for Service interrupt by reading the Host Status (HSTAT) SFR to determine the source of the interrupt. The Host then reads the Threshold number of bytes from the FIFO. The internal CPU may continue to write to the FIFO during the Host read of the FIFO Output channel.

---

**Table 3. UPI-452 External Address Decoding**

<table>
<thead>
<tr>
<th>DACK</th>
<th>CS</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>READ</th>
<th>WRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>No Operation</td>
<td>No Operation</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Data or DMA from Output FIFO Channel</td>
<td>Data or DMA to Input FIFO Channel</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Data Stream Command from Output FIFO Channel</td>
<td>Data Stream Command to Input FIFO Channel</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Host Status SFR Read</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Host Control SFR Read</td>
<td>Host Control SFR Write</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Immediate Command SFR Read</td>
<td>Immediate Command SFR Write</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>DMA Data from Output FIFO Channel</td>
<td>DMA Data to Input FIFO Channel</td>
</tr>
</tbody>
</table>
Data Stream Commands may be written to the Output FIFO channel at any time during a write of data bytes. The write instruction need only specify the Command Out (COUT) SFR in the direct register instruction used. Immediate Commands may also be written at any time to the Immediate Command OUT (IMOUT) SFR. The Host reads Immediate Commands from the Immediate Command OUT (IMOUT).

The internal CPU can determine the number of bytes to write to the FIFO channel in one of three ways. The first, and most efficient, is by utilizing the internal DMA processor which will automatically manage the writing of data to avoid Underrun or Overrun Errors. The second is for the internal CPU to read the Output FIFO channels Read and Write Pointers and compare their values to determine the available space. The third method for determining the available FIFO space is to always write the programmed channel size number of bytes to the Output FIFO. This method would use the Overrun Error flag and interrupt to halt FIFO writing whenever the available space was less than the channel size. The interrupt service routine could read the channel pointers to determine or monitor the available channel space. The time required for the internal CPU to write data to the Output FIFO channel is a function of the individual instruction cycle time and the number of bytes to be written.

**Host Read from the FIFO**

The Host reads data or Data Stream Commands (DSC) from the FIFO in response to the Host Status (HSTAT) SFR flags and interrupts, if enabled. All Host read operations access the same UPI-452 internal I/O Buffer Latch. At the end of the previous Host FIFO read cycle a byte is loaded from the FIFO into the I/O Buffer Latch and Host Status (HSTAT) SFR bit 5 is set or cleared (1 = DSC, 0 = data) to reflect the state of the byte's FIFO ninth bit. If the FIFO ninth bit is set (= 1) indicating a DSC, an interrupt is generated to the external Host via INTRQ pin or INTRQIN/INTRQOUT pins as determined by Host Control (HCON) SFR bit 1. The Host then reads the Host Status (HSTAT) SFR to determine the source of the interrupt.

The most efficient Host read operation of the FIFO Output channel is through the use of Host DMA. The UPI-452 fully supports external DMA handshaking. The MODE and Host Control SFRs control the configuration of UPI-452 Host DMA handshake outputs. If Host DMA is used the Threshold Request for Service interrupt asserts the UPI-452 DMA Request (DRQOUT) output. The Host DMA processor acknowledges with DACK which acts as a chip select of the FIFO channels. The DMA transfer would stop when either the threshold byte count had been read, as programmed in the Host DMA processor, or when the DRQOUT output is brought inactive by the UPI-452.

**INPUT CHANNEL**

This section covers the data path from the HOST to the internal CPU or internal DMA processor. The details of Data Stream Command or Immediate Command processing during internal DMA operations are covered in the DMA section below.

**Host Write to the FIFO**

The Host writes data and Data Stream Commands into the FIFO through the FIFO IN (FIN) and Command IN (CIN) SFRs. When a Threshold number of bytes has been read out of the Input FIFO channel by the internal CPU, the Host Status SFR Input FIFO Request for Service bit is set and an interrupt, if enabled, is generated to the Host. The Input FIFO Threshold interrupt tells the Host that it may write the next block of data into the FIFO. Either the INTRQ or DRQIN/INTRQIN output pins can be used for this interrupt as determined by the MODE and Host Control (HCON) SFR settings. The Host may continue to write to the FIFO Input channel during the internal CPU read of the FIFO. Data Stream Commands may be written to the FIFO Input channel at any time during a write of data bytes. Immediate Commands may also be written at any time to the Immediate Command IN (IMIN) SFR.
The Host also has three methods for determining the available FIFO space. Two are essentially identical to that of the internal CPU. They involve reading the FIFO Input channel pointers and using the Host Status SFR Underrun and Overrun Error flags and Request for Service interrupts these would generate, if enabled in combination. The third involves using the UPI-452 Host DMA controller handshake signals and the programmed Input FIFO Threshold. The Host would receive a Request for Service interrupt when an Input FIFO channel has a Threshold number of bytes able to be written by the Host. The Host service routine would then write the Threshold number of bytes to the FIFO.

If a Host DMA is used to write to the FIFO Input channel, the Threshold Request for Service interrupt could assert the UPI-452 DRQIN output. The Host DMA processor would assert DACK and the FIFO write would be completed by Host the DMA processor. The DMA transfer would stop when either the Threshold byte count had been written or the DRQIN output was removed by the UPI-452. Additional details on Host and internal DMA operation is given below.

Internal Read of the FIFO

At the end of an internal CPU read cycle a byte is loaded from the FIFO buffer into the FIFO IN/Command IN SFR and Slave Status (SSTAT) SFR bit 1 is set or cleared (1 = data, 0 = DSC) to reflect the state of the FIFO ninth bit. If the byte is a DSC, the FIFO ninth bit is set (= 1) and an interrupt is generated, if enabled, to the Internal CPU. The internal CPU then reads the Slave Status (SSTAT) SFR to determine the source of the interrupt.

Immediate Commands are written by the Host and read by the internal CPU through the Immediate Command IN (IMIN) SFR. Once written, an Immediate Command sets the Slave Status (SSTAT) SFR flag bit and generates an interrupt, if enabled, to the internal CPU. In response to the interrupt the internal CPU reads the Slave Status (SSTAT) SFR to determine the source of the interrupt and service the Immediate Command.

FIFO INPUT/OUTPUT CHANNEL SIZE

Host

The Host does not have direct control of the FIFO Input or Output channel sizes or configuration. The Host can, however, issue Data Stream Commands or Immediate Commands to the UPI-452 to reconfigure the FIFO interface by invoking FIFO DMA Freeze Mode. The Data Stream Command or Immediate Command would be a vector to a service routine which performs the specific reconfiguration.

UPI-452 Internal

The default power-on reset FIFO channel sizes are listed in the “Initialization” section and can be set only by the internal CPU during FIFO DMA Freeze Mode. The FIFO channel size is selected to achieve the optimum application performance. The entire 128 byte FIFO can be allocated to either the Input or Output channel. In this case the other channel consists of a single SFR; FIFO IN/Command IN or FIFO OUT/Command OUT SFR. Figure 4 shows a FIFO division with a portion devoted to each channel. Figure 5 shows a FIFO configuration with all 128 bytes assigned to the Output channel.

The FIFO channel Threshold feature allows the user to match the FIFO channel size and the performance of the internal and Host data transfer rates. The programmed Threshold provides an elasticity to the data transfer operation. An example is if the Host FIFO...
data transfer rate is twice as fast as the internal FIFO DMA data transfer rate. In this example the FIFO Input channel size is programmed to be 64 bytes and the Input channel Threshold is programmed to be 20 bytes. The Host writes the first 64 bytes to the Input FIFO. When the internal DMA processor has read 20 bytes the Threshold interrupt, or DMA request (DRQIN), is generated to signal the Host to begin writing more data to the Input FIFO channel. The internal DMA processor continues to read data from the Input channel as the Host, or Host DMA processor, writes to the FIFO. The Host can write 40 bytes to the FIFO Input channels in the time it takes for the internal DMA processor to read 20 more bytes from it. This will keep both the Host and internal DMA operating at their maximum rates without forcing one to wait for the other.

Two methods of managing the FIFO size are possible; fixed and variable channel size. A fixed channel size is one where the channel is configured at initialization and remains unchanged throughout program execution. In a variable FIFO channel size, the configuration is changed dynamically to meet the data transmission requirements as needed. An example of a variable channel size application is the mass storage subsystem described earlier. To meet the demands of a large data block transfer the FIFO size could be fully allocated to the Input or Output channel as needed. The Thresholds are also reprogrammed to match the respective data transfer rates.

An example of a fixed channel size application might be one which uses the UPI-452 to directly control a series of stepper motors. The UPI-452 manages the motor operation and status as required. This would include pulse train, acceleration, deceleration and feedback. The Host transmits motor commands to the UPI-452 in blocks of 6–10 bytes. Each block of motor command data is preceded by a command to the UPI-452 which selects a specific motor. The UPI-452 transmits blocks of data to the Host which provides motor and overall system status. The data and embedded commands structure, indicating the specific motor, is the same. In this example the default 64 bytes per channel might be adequate for both channels.

**INTERRUPT RESPONSE TIMING**

Interrupts enable the Host UPI-452 FIFO buffer interface and the internal CPU FIFO buffer interface to operate with a minimum of overhead on the respective CPU. Each CPU is “interrupted” to service the FIFO on an as needed basis only. In configuring the FIFO buffer Thresholds and choosing the appropriate internal DMA Mode the user must take into account the interrupt response time for both CPUs. These response times will affect the DMA transfer rates for each channel. By choosing FIFO channel Thresholds which reflect both the respective DMA transfer rate and the interrupt response time the user will achieve the maximum data throughput and system bus decoupling. This in turn will mean the overall available system bus bandwidth will increase.

The following section describes the FIFO interrupt interface to the Host and internal CPU. It also describes an analysis of sample interrupt response times for the Host and UPI-452 internal CPU. These equations and the example times shown are then used in the DMA section to further analyze an optimum Host UPI-452 interface.

**HOST**

Interrupts to the Host processor are supported by the three UPI-452 output pins; INTRQ, DRQIN/INTRQIN and DRQOUT/INTRQOUT. INTRQ is a general purpose Request For Service interrupt output. DRQIN/INTRQIN and DRQOUT/INRQOUT pins are multiplexed to provide two special “Request for Service” FIFO interrupt request lines when DMA is disabled. A FIFO Input or Output channel Request for Service interrupt is generated based upon the value programmed in the respective channel’s Threshold SFRs; Input Threshold (ITHR), and Output Threshold
itSFRs. Additional interrupts are provided for FIFO Underrun and Overrun Errors, Data Stream Commands, and Immediate Commands. Table 4 lists the eight UPI-452 interrupt sources as they appear in the HSTAT SFR to the Host processor.

### Table 4. UPI-452 to Host Interrupt Sources

<table>
<thead>
<tr>
<th>HSTAT SFR Bit</th>
<th>Interrupt Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>HST7</td>
<td>Output FIFO Underrun Error</td>
</tr>
<tr>
<td>HST6</td>
<td>Immediate Command Out SFR Status</td>
</tr>
<tr>
<td>HST5</td>
<td>Data Stream Command/Data at Output FIFO Status</td>
</tr>
<tr>
<td>HST4</td>
<td>Output FIFO Request for Service Status</td>
</tr>
<tr>
<td>HST3</td>
<td>Input FIFO Overrun Error Condition</td>
</tr>
<tr>
<td>HST2</td>
<td>Immediate Command in SFR Status</td>
</tr>
<tr>
<td>HST1</td>
<td>FIFO DMA Freeze/Normal Mode Status</td>
</tr>
<tr>
<td>HST0</td>
<td>Input FIFO Request for Service</td>
</tr>
</tbody>
</table>

The interrupt response time required by the Host processor is application and system specific. In general, a typical sequence of Host interrupt response events and the approximate times associated with each are listed in Equation 1.

The example assumes the hardware configuration shown in Figure 1, iAPX 286/UPi-452 Block Diagram, with an 8259A Programmable Interrupt Controller. The timing analysis in Equation 1 also assumes the following: no other interrupt is either in process or pending, nor is the 286 in a LOCK condition. The current instruction completion time is 8 clock cycles (800 ns @ 10 MHz), or 4 bus cycles. The interrupt service routine first executes a PUSH instruction, PUSH All General Registers, to save all iAPX 286 internal registers. This requires 19 clocks (or 2.0 μs @ 10 MHz), or 10 bus cycles (rounded to complete bus cycle). The next service routine instruction reads the UPI-452 Host Status SFR to determine the interrupt source.

It is important to note that any UPI-452 INTRQ interrupt service routine should ALWAYS mask for the Freeze Mode bit first. This will insure that Freeze Mode always has the highest priority. This will also save the time required to mask for bits which are forced inactive during Freeze Mode, before checking the Freeze Mode bit. Access to the FIFO channels by the Host is inhibited during Freeze Mode. Freeze Mode is covered in more detail below.

To initiate the interrupt the UPI-452 activates the INTRQ output. The interrupt acknowledge sequence requires two bus cycles, 400 ns (10 MHz iAPX 286), for the two INTA pulse sequence.

### Equation 1. Host Interrupt Response Time

<table>
<thead>
<tr>
<th>Action</th>
<th>Time</th>
<th>Bus Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current instruction execution</td>
<td>800 ns</td>
<td>4</td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTA sequence</td>
<td>400 ns</td>
<td>2</td>
</tr>
<tr>
<td>Interrupt service routine (time to host first READ of UPI-452)</td>
<td>2000 ns</td>
<td>10</td>
</tr>
<tr>
<td>Total Interrupt Response Time</td>
<td>2.3 μs</td>
<td>16</td>
</tr>
</tbody>
</table>

**NOTE:**
10 MHz iAPX 286 bus cycle, 200 ns each

### UPI-452 Internal

The internal CPU FIFO interrupt interface is essentially identical to that of the Host to the FIFO. Three internal interrupt sources support the FIFO operation; FIFO-Slave bus Interface Buffer, DMA Channel 0 and DMA Channel 1 Requests. These interrupts provide a maximum decoupling of the FIFO buffer and the internal CPU. The four different internal DMA Modes available add flexibility to the interface.

The FIFO-Slave Bus Interface interrupt response is also similar to the Host response to an INTRQ Request for Service interrupt. The internal CPU responds to the interrupt by reading the Slave Status (SSTAT) SFR to determine the source of the interrupt. This allows the user to prioritize the Slave Status flag response to meet the users application needs.

The internal interrupt response time is dependent on the current instruction execution, whether the interrupt is enabled, and the interrupt priority. In general, to finish execution of the current instruction, respond to the interrupt request, push the Program Counter (PC) and vector to the first instruction of the interrupt service routine requires from 38 to 86 oscillator periods (2.38 to 5.38 μs @ 16 MHz). If the interrupt is due to an Immediate Command or DSC, additional time is required to read the Immediate Command or DSC SFR and vector to the appropriate service routine. This means two service routines back to back. One service routine to read the Slave Status (SSTAT) SFR to determine the source of the Request for Service interrupt, and second the service routine pointed to by the Immediate Command or DSC byte read from the respective SFR.
DMA

DMA is the fastest and most efficient way for the Host or internal CPU to communicate with the FIFO buffer. The UPI-452 provides support for both of these DMA paths. The two DMA paths and operations are fully independent of each other and can function simultaneously. While the Host DMA processor is performing a DMA transfer to or from the FIFO, the UPI-452 internal DMA processor can be doing the same.

Below are descriptions of both the Host and internal DMA operations. Both DMA paths can operate asynchronously and at different transfer rates. In order to make the most of this simultaneous asynchronous operation it is necessary to calculate the two transfer rates and accurately match their operations. Matching the different transfer rates is done by a combination of accurately programmed FIFO channel size and channel Thresholds. This provides the maximum Host and internal CPU to FIFO buffer interface decoupling. Below is a description of each of the two DMA operations and sample calculations for determining transfer rates. The next section of this application note, “Interface Latency”, details the considerations involved in analyzing effective transfer rates when the overhead associated with each transfer is considered.

HOST FIFO DMA

DMA transfers between the Host and UPI-452 FIFO buffer are controlled by the Host CPU’s DMA controller, and is independent of the UPI-452’s internal two channel DMA processor. The UPI-452’s internal DMA processor supports data transfers between the UPI-452 internal RAM, external RAM (via the Local Expansion Bus) and the various Special Function Registers including the FIFO Input and Output channel SFRs.

The maximum DMA transfer rate is achieved by the minimum DMA transfer cycle time to accomplish a source to destination move. The minimum Host UPI-452 FIFO DMA cycle time possible is determined by the READ and WRITE pulse widths, UPI-452 command recovery times in relation to the DMA transfer timing and DMA controller transfer mode used. Table 5 shows the relationship between the iAPX-286, iAPX-186 and UPI-452 for various DMA as well as non-DMA byte by byte transfer modes versus processor frequencies.

Table 5. Host UPI-452 Data Transfer Performance

<table>
<thead>
<tr>
<th>Processor &amp; Speed</th>
<th>Wait States: Back to Back READ/ WRITE’s</th>
<th>DMA: Single Cycle</th>
<th>Two Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>iAPX-186* 8 MHz</td>
<td>0</td>
<td>N/A*</td>
<td>0</td>
</tr>
<tr>
<td>10 MHz</td>
<td>0</td>
<td>N/A*</td>
<td>0</td>
</tr>
<tr>
<td>12.5 MHz</td>
<td>1</td>
<td>N/A*</td>
<td>0</td>
</tr>
<tr>
<td>iAPX-286** 6 MHz</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8 MHz</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10 MHz</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTES:
* iAPX 186 On-chip DMA processor is two cycle operation only.
** iAPX 286 assumes 82258 ADMA (or other DMA) running 286 bus cycles at 286 clock rate.

In this application note system example, shown in Figure 1, DMA operation is assumed to be two bus cycle I/O to memory or memory to I/O. Two cycle DMA consists of a fetch bus cycle from the source and a store bus cycle to the destination. The data is stored in the DMA controller’s registers before being sent to the destination. Single cycle DMA transfers involve a simultaneous fetch from the source and store to the destination. As the most common method of I/O-memory DMA operation, two cycle DMA transfers are the focus of this application note analysis. Equation 2 illustrates a calculation of the overall transfer rate between the FIFO buffer and external Host for a maximum FIFO size transfer. The equation does not account for the latency of initiating the DMA transfer.

Equation 2. Host FIFO DMA Transfer Rate—Input or Output Channel

\[
\text{Cycle DMA Transfer-I/O (UPI-452) to System Memory} = \text{FIFO channel size} \times (\text{DMA READ/WRITE FIFO time + DMA WRITE/READ Memory Time}) = 128 \text{ bytes} \times (200 \text{ ns} + 200 \text{ ns}) = 51.2 \mu\text{s} = 256 \text{ bus cycles}\]

NOTES:
* 10 MHz iAPX 286, 200 ns bus cycles.

The UPI-452 design is optimized for high speed DMA transfers between the Host and the FIFO buffer. The
UPI-452 internal FIFO buffer control logic provides the necessary synchronization of the external Host event and the internal CPU machine cycle during UPI-452 RD/WR accesses. This internal synchronization is addressed by the TCC AC specification of the UPI-452 shown in Figure 6. TCC is the time from the leading or trailing edge of a UPI-452 RD/WR to the same edge of the next UPI-452 RD/WR. The TCC time is effectively another way of measuring the system bus cycle time with reference to UPI-452 accesses.

In the iAPX-286 10 MHz system depicted in this application note the bus cycle time is 200 ns. Alternate cycle accesses of the UPI-452 during two cycle DMA operation yields a TCC time of 400 ns which is more than the TCC minimum time of 375 ns. Back to back Host UPI-452 READ/WRITE accesses may require wait states as shown in Table 5. The difference between 10 MHz iAPX-186 and 10 MHz iAPX 286 required wait states is due to the number of clock cycles in the respective bus cycle timings. The four clocks in a 10 MHz iAPX 186 bus cycle means a minimum TCC time of 400 ns versus 200 ns for a 10 MHz iAPX 286 with two clock cycle zero wait state bus cycle.

DMA handshaking between the Host DMA controller and the UPI-452 is supported by three pins on the UPI-452: DRQIN/INTRQIN, DRQOUT/INTRQOUT, and DACK. The DRQIN/INTRQIN and DRQOUT/INTRQOUT outputs are two multiplexed DMA or interrupt request pins. The function of these pins is controlled by MODE SFR bit 6 (MD6). DRQIN and DRQOUT provide a direct interface to the Host system DMA controller (see Figure 1). In response to a DRQIN or DRQOUT request, the Host DMA controller initiates control of the system bus using HLD/HLDA. The FIFO Input or Output channel transfer is accomplished with a minimum of Host overhead and system bus bandwidth.

The third handshake signal pin is DACK which is used as a chip select during DMA data transfers. The UPI-452 Host READ and WRITE input signals select the respective Input and Output FIFO channel during DMA transfers. The CS and address lines provide DMA acknowledge for processors with onboard DMA controllers which do not generate a DACK signal.

The iAPX 286 Block I/O Instructions provide an alternative to two cycle DMA data transfers with approximately the same data rate. The String Input and Output instructions (INS & OUTS) when combined with the Repeat (REP) prefix, modifies INS and OUTS to provide a means of transferring blocks of data between I/O and Memory. The data transfer rate using REP INS/OUTS instructions is calculated in the same way as two cycle DMA transfer times. Each READ or WRITE would be 200 ns in a 10 MHz iAPX 286 system. The maximum transfer rate possible is 2.5 MBytes/second. The Block I/O FIFO data transfer calculation is the same as that shown in Equation 2 for two cycle DMA data transfers including TCC timing effects.

**FIFO Data Structure and Host DMA**

During a Host DMA write to the FIFO, if a DSC is to be written, the DMA transfer is stopped, the DSC is written and the DMA restarted. During a Host DMA read from the FIFO, if a DSC is loaded into the I/O Buffer Latch the DMA request, DRQOUT, will be deactivated (see Figure 2 above). The Host Status (HSTAT) SFR Data Stream Command bit is set and the INTRQ interrupt output goes active, if enabled. The Host responds to the interrupt as described above.
Once INTRQ is deactivated and the DSC has been read by the Host, the DMA request, DRQOUT, is reasserted by the UPI-452. The DMA request then remains active until the transfer is complete or another DSC is loaded into the I/O Buffer Latch.

An Immediate Command written by the internal CPU during a Host DMA FIFO transfer also causes the Host Status flag and INTRQ to go active if enabled. In this case the Immediate Command would not terminate the DMA transfer unless terminated by the Host. The INTRQ line remains active until the Host reads the Host Status (HSTAT) SFR to determine the source of the interrupt.

The net effect of a Data Stream Command (DSC) on DMA data transfer rates is to add an additional factor to the data transfer rate equation. This added factor is shown in Equation 3. An Immediate Command has the same effect on the data transfer rate if the Immediate Command interrupt is recognized by the Host during a DMA transfer. If the DMA transfer is completed before the Immediate Command interrupt is recognized, the effect on the DMA transfer rate depends on whether the block being transmitted is larger than the FIFO channel size. If the block is larger than the programmed FIFO channel size the transfer rate depends on whether the Immediate Command flag or interrupt is recognized between partial block transfers.

The FIFO configuration shown in Equation 3 is arbitrary since there is no way of predicting the size relative to when a DSC would be loaded into the I/O Buffer Latch. The Host DMA rate shown is for a UPI-452 (Memory Mapped or I/O) to 286 System Memory transfer as described earlier. The equations do not account for the latency of initiating the DMA transfer.

**Equation 3. Minimum host FIFO DMA Transfer Rate Including Data Stream Command(s)**

Minimum Host/FIFO DMA Transfer Rate w/ DSC

\[
\text{Minimum Host/FIFO DMA Transfer Rate} = \text{FIFO size} \times \text{Host DMA 2 cycle time transfer rate} + \text{iAPX 286 interrupt response time (Eq. #1)}
\]

\[
= (32 \text{ bytes} \times (200 \text{ ns} + 200 \text{ ns})) + 2.3 \mu\text{s}
\]

\[
= 15.1 \mu\text{s}
\]

\[
= 75.5 \text{ bus cycles (@10 MHz iAPX286, 200 ns bus cycle)}
\]

**UPI-452 INTERNAL DMA PROCESSOR**

The two identical internal DMA channels allow high speed data transfers from one UPI-452 writable memory space to another. The following UPI-452 memory spaces can be used with internal DMA channels:

- Internal Data Memory (RAM)
- External Data Memory (RAM)
- Special Function Registers (SFR)

The FIFO can be accessed during internal DMA operations by specifying the FIFO IN (FIN) SFR as the DMA Source Address (SAR) or the FIFO OUT (FOUT) SFR as the Destination Address (DAR). Table 6 lists the four types of internal DMA transfers and their respective timings.

**Table 6. UPI-452 Internal DMA Controller Cycle Timings**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Machine Cycles**</th>
<th>@12 MHz</th>
<th>@16 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Data Mem. or SFR</td>
<td>Internal Data Mem. or SFR</td>
<td>1</td>
<td>1 \mu s</td>
<td>750 ns</td>
</tr>
<tr>
<td>Internal Data Mem. or SFR</td>
<td>External Data Mem.</td>
<td>1</td>
<td>1 \mu s</td>
<td>750 ns</td>
</tr>
<tr>
<td>External Data Mem. *External Data Memory</td>
<td>Internal Data Mem. or SFR</td>
<td>1</td>
<td>1 \mu s</td>
<td>750 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2 \mu s</td>
<td>1.5 \mu s</td>
</tr>
</tbody>
</table>

**NOTES:**

* *External Data Memory DMA transfer applies to UPI-452 Local Bus only.
**MSC-51 Machine cycle = 12 clock cycles (TCLCL).
FIFO Data Structure and Internal DMA

The effect of Data Stream Commands and Immediate Commands on the internal DMA transfers is essentially the same as the effect on Host FIFO DMA transfers. Recognition also depends upon the programmed DMA Mode, the interrupts enabled, and their priorities. The net internal effect is the same for each possible internal case. The time required to respond to the Immediate or Data Stream Command is a function of the instruction time required. This must be calculated by the user based on the instruction cycle time given in the MSC-51 Instruction Set description in the Intel Microcontroller Handbook.

It is important to note that the internal DMA processor modes and the internal FIFO logic work together to automatically manage internal DMA transfers as data moves into and out of the FIFO. The two most appropriate internal DMA processor modes for the FIFO are FIFO Demand Mode and FIFO Alternate Cycle Mode. In FIFO Demand Mode, once the correct Slave Control and DMA Mode bits are set, the internal Input FIFO channel DMA transfer occurs whenever the Slave Control Input FIFO Request for Service flag is set. The DMA transfer continues until the flag is cleared or when the Input FIFO Read Pointer SFR (IRPR) equals zero. If data continues to be entered by the Host, the internal DMA continues until an internal interrupt of higher priority, if enabled, interrupts the DMA transfer, the internal DMA byte count reaches zero or until the Input FIFO Read Pointer equals zero. A complete description of interrupts and DMA Modes can be found in the UPI-452 Data Sheet.

DMA Modes

The internal DMA processor has four modes of operation. Each DMA channel is software programmable to operate in either Block Mode or Demand Mode. Demand Mode may be further programmed to operate in Burst or Alternate Cycle Mode. Burst Mode causes the internal processor to halt its execution and dedicate its resources exclusively to the DMA transfer. Alternate Cycle Mode causes DMA cycles and instruction cycles to occur alternately. A detailed description of each DMA Mode can be found in the UPI-452 Data Sheet.

INTERFACE LATENCY

The interface latency is the time required to accommodate all of the overhead associated with an individual data transfer. Data transfer rates between the Host system and UPI-452 FIFO, with a block size less than or equal to the programmed FIFO channel size, are calculated using the Host system DMA rate. (see Host DMA description above). In this case, the entire block could be transferred in one operation. The total latency is the time required to accomplish the block DMA transfer, the interrupt response or poll of the Host Status SFR response time, and the time required to initiate the Host DMA processor.

A DMA transfer between the Host and UPI-452 FIFO with a block size greater than the programmed FIFO channel size introduces additional overhead. This additional overhead is from three sources; first, is the time to actually perform the DMA transfer. Second, the overhead of initializing the DMA processor, third, the handshaking between each FIFO block required to transfer the entire data block. This could be time to wait for the FIFO to be emptied and/or the interrupt response time to restart the DMA transfer of the next portion of the block. A fourth component may also be the time required to respond to Underrun and Overrun FIFO Errors.

Table 7 shows six typical FIFO Input/Output channel sizes and the Host DMA transfers times for each. The timings shown reflect a 10 MHz system bus two cycle I/O to Memory DMA transfer rate of 2.5 MBytes/second as shown in Equation 1. The times given would be the same for iAPX 286 I/O block move instructions REP INS and REP OUTS as described earlier.

<table>
<thead>
<tr>
<th>FIFO Size:</th>
<th>32</th>
<th>43</th>
<th>64</th>
<th>85</th>
<th>96</th>
<th>128</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full or Empty</td>
<td>¼</td>
<td>½</td>
<td>¾</td>
<td>¾</td>
<td>¾</td>
<td>Full or Empty</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>12.8</td>
<td>17.2</td>
<td>25.6</td>
<td>34.0</td>
<td>38.4</td>
<td>51.2</td>
<td>μs</td>
</tr>
</tbody>
</table>

Table 8 shows six typical FIFO Input/Output channel sizes and the internal DMA processor data transfers times for each. The timings shown are for a UPI-452 single cycle Burst Mode transfer at 16 MHz or 750 ns per machine cycle in or out of the FIFO channels. The
machine cycle time is that of the MSC-51 CPU; 6 states, 2 XTL2 clock cycles each or 12 clock cycles per machine cycle. Details on the MSC-51 machine cycle timings and operation may be found in the Intel Microcontroller Handbook.

Table 8. UPI-452 Internal DMA FIFO Data Transfer Times

<table>
<thead>
<tr>
<th>FIFO Size:</th>
<th>32</th>
<th>43</th>
<th>64</th>
<th>85</th>
<th>96</th>
<th>128 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full or Empty</td>
<td>⅓</td>
<td>⅓</td>
<td>⅔</td>
<td>⅔</td>
<td>⅔</td>
<td>Full or Empty</td>
</tr>
<tr>
<td>Time</td>
<td>24.0</td>
<td>32.3</td>
<td>48.0</td>
<td>64.6</td>
<td>72.0</td>
<td>96.0 μs</td>
</tr>
</tbody>
</table>

A larger than programmed FIFO channel size data block internal DMA transfer requires internal arbitration. The UPI-452 provides a variety of features which support arbitration between the two internal DMA channels and the FIFO. An example is the internal DMA processor FIFO Demand Mode described above. FIFO Demand Mode DMA transfers occur continuously until the Slave Status Request for Service Flag is deactivated. Demand Mode is especially useful for continuous data transfers requiring immediate attention. FIFO Alternate Cycle Mode provides for interleaving DMA transfers and instruction cycles to achieve a maximum of software flexibility. Both internal DMA channels can be used simultaneously to provide continuous transfer for both Input and Output FIFO channels.

Byte by byte transfers between the FIFO and internal CPU timing is a function of the specific instruction cycle time. Of the 111 MCS-51 instructions, 64 require 12 clock cycles, 45 require 24 clock cycles and 2 require 48 clock cycles. Most instructions involving SFRs are 24 clock cycles except accumulator (for example, MOV direct, A) or logical operations (ANL direct, A). Typical instruction and their timings are shown in Table 9.

Oscillator Period: @ 12 MHz = 83.3 ns
@ 16 MHz = 62.5 ns

Table 9. Typical Instruction Cycle Timings

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Oscillator Periods</th>
<th>@12 MHz</th>
<th>@16 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV direct†, A</td>
<td>12</td>
<td>1 μs</td>
<td>750 ns</td>
</tr>
<tr>
<td>MOV direct, direct</td>
<td>24</td>
<td>2 μs</td>
<td>1.5 μs</td>
</tr>
</tbody>
</table>

NOTE: † Direct = 8-bit internal data locations address. This could be an Internal Data RAM location (0–255) or a SFR [i.e., 1/0 port, control register, etc.]

Byte by byte FIFO data transfers introduce an additional overhead factor not found in internal DMA operations. This factor is the FIFO block size to be transferred; the number of empty locations in the Output channel, or the number of bytes in the Input FIFO channel. As described above in the FIFO Data Structure section, the block size would have to be determined by reading the channel read and write pointer and calculating the space available. Another alternative uses the FIFO Overrun and Underrun Error flags to manage the transfers by accepting error flags. In either case the instructions needed have a significant impact on the internal FIFO data transfer rate latency equation.

A typical effective internal FIFO channel transfer rate using internal DMA is shown in Equation 4. Equation 5 shows the latency using byte by byte transfers with an arbitrary factor added for internal CPU block size calculation. These two equations contrast the effective transfer rates when using internal DMA versus individual instructions to transfer 128 bytes. The effective transfer rate is approximately four times as fast using DMA versus using individual instructions (96 μs with DMA versus 492 μs non-DMA).

Equation 4. Effective Internal FIFO Transfer Time Using Internal DMA

Effective Internal FIFO Transfer Rate with DMA
= FIFO channel size * Internal DMA Burst Mode Single Cycle DMA Time
= 128 Bytes * 750 ns
= 96 μs

Equation 5. Effective FIFO Transfer Time Using Individual Instructions

Effective Internal FIFO Transfer Rate without DMA
= FIFO channel size * Instruction Cycle Time + Block size calculation Time
= 128 Bytes * (24 oscillator periods @ 16 MHz) + 20 instructions (24 oscillator period each @ 16 MHz)
= 128 * 1.5 μs + 300 μs
= 492 μs

FIFO DMA FREEZE MODE INTERFACE

FIFO DMA Freeze Mode provides a means of locking the Host out of the FIFO Input and Output channels. FIFO DMA Freeze Mode can be invoked for a variety of reasons, for example, to reconfigure the UPI-452 Local Expansion Bus, or change the baud rate on the serial channel. The primary reason the FIFO DMA Freeze Mode is provided is to ensure that the Host does not read from or write to the FIFO while the FIFO interface is being altered. ONLY the internal CPU has the capability of altering the FIFO Special Function Registers, and these SFRs can ONLY be altered during FIFO DMA Freeze Mode. FIFO DMA Freeze Mode inhibits Host access of the FIFO while the internal CPU reconfigures the FIFO.
FIFO DMA Freeze Mode should not be arbitrarily invoked while the UPI-452 is in normal operation. Because the external CPU runs asynchronously to the internal CPU, invoking freeze mode without first properly resolving the FIFO Host interface may have serious consequences. Freeze Mode may be invoked only by the internal CPU.

The internal CPU invokes Freeze Mode by setting bit 3 of the Slave Control SFR (SC3). This automatically forces the Slave and Host Status SFR FIFO DMA Freeze Mode to In Progress (SSTAT SST5 = 0, HSTAT SFR HST1 = 1). INTRQ goes active, if enabled by MODE SFR bit 4, whenever FIFO DMA Freeze Mode is invoked to notify the Host. The Host reads the Host Status SFR to determine the source of the interrupt. INTRQ and the Slave and Host Status FIFO DMA Freeze Mode bits are reset by the Host READ of the Host Status SFR.

During FIFO DMA Freeze Mode the Host has access to the Host Status and Control SFRs. All other Host FIFO interface access is inhibited. Table 10 lists the FIFO DMA Freeze Mode status of all slave bus interface Special Function Registers. The internal DMA processor is disabled during FIFO DMA Freeze Mode and the internal CPU has write access to all of the FIFO control SFRs (Table 11).

If FIFO DMA Freeze Mode is invoked without stopping the host, only the last two bytes of data written into or read from the FIFO will be valid. The timing diagram for disabling the FIFO module to the external Host interface is illustrated in Figure 7. Due to this synchronization sequence, the UPI-452 might not go into FIFO DMA Freeze Mode immediately after the Slave Control SFR FIFO 7 DMA Freeze Mode bit (SC3) is set = 0. A special bit in the Slave Status SFR (SST5) is provided to indicate the status of the FIFO DMA Freeze Mode. The FIFO DMA Freeze Mode operations described in this section are only valid after SST5 is cleared.

Either the Host or internal CPU can request FIFO DMA Freeze Mode. The first step is to issue an Immediate Command indicating that FIFO DMA Freeze Mode will be invoked. Upon receiving the Immediate Command, the external CPU should complete servicing all pending interrupts and DMA requests, then send an Immediate Command back to the internal CPU acknowledging the FIFO DMA Freeze Mode request. After issuing the first Immediate Command, the internal CPU should not perform any action on the FIFO until FIFO DMA Freeze Mode is invoked. The handshaking is the same in reverse if the HOST CPU initiates FIFO DMA Freeze Mode.

After the slave bus interface is frozen, the internal CPU can perform the operations listed below on the FIFO Special Function Registers. These operations are allowed only during FIFO DMA Freeze Mode. Table 11 summarizes the characteristics of all the FIFO Special Function Registers during Normal and FIFO DMA Freeze Modes.

For FIFO
Reconfiguration
1. Changing the Channel Boundary Pointer SFR.

To Enhance the testability
3. Writing to the read and write pointers of the Input and Output FIFO's.

4. Writing to and reading the Host Control SFRs.

5. Controlling some bits of Host and Slave Status SFRs.

6. Reading the Immediate Command Out SFR and Writing to the Immediate Command in SFR.

NOTE:
Timing Diagram of disabling of FIFO Module—External Host Interface.

Figure 7. Disabling FIFO to Host Slave Interface Timing Diagram
The sequence of events for invoking FIFO DMA Freeze Mode are listed in Figure 8.

1. Immediate Command to request FIFO DMA Freeze Mode (interrupt)
2. Host/internal CPU interrupt response/service
3. Host/internal CPU clear/service all pending interrupts and FIFO data
4. Internal CPU sets Slave Control (SLCON) FIFO DMA
   Freeze Mode bit = 0, FIFO DMA Freeze Mode, Host Status SFR FIFO DMA Freeze Mode Status bit = 1, INTRQ active (high)
5. Host READ Host Status SFR
6. Internal CPU reconfigures FIFO SFRs
7. Internal CPU resets Slave Control (SLCON) FIFO DMA
   Freeze Mode bit = 1, Normal Mode, Host Status FIFO DMA Freeze Mode Status bit = 0.
8. Internal CPU issues Immediate Command to Host indicating that FIFO DMA Freeze Mode is complete
   or
   Host polls Host Status SFR FIFO DMA Freeze Mode bit to determine end of reconfiguration

Figure 8. Sequence of Events to Invoke FIFO DMA Freeze Mode

EXAMPLE CONFIGURATION

An example of the time required to reconfigure the FIFO 180 degrees, for example from 128 bytes Input to 128 bytes Output, is shown in Figure 9. The example approximates the time based on several assumptions;
1. The FIFO Input channel is full-128 bytes of data
2. Output FIFO channel is empty-1 byte
3. No Data Stream Commands in the FIFO.
4. The Immediate Command interrupt is responded to immediately—highest priority—by Host and internal CPU.
5. Respective interrupt response times
   a. Host (Equation 3 above) = approximately 1.6 µs
   b. Internal CPU is 86 oscillator periods or approximately 5.38 µs worst case.

<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Command from Host to UPI-452 to request FIFO DMA Freeze Mode (IAPX286 WRITE)</td>
<td>0.30 µs</td>
</tr>
<tr>
<td>Internal CPU interrupt response/service</td>
<td>5.38 µs</td>
</tr>
<tr>
<td>Internal CPU clears FIFO-128 bytes DMA</td>
<td>96.00 µs</td>
</tr>
<tr>
<td>Internal CPU sets Slave Control Freeze Mode bit</td>
<td>0.75 µs</td>
</tr>
<tr>
<td>Immediate Command to Host-Freeze Mode in progress Host Immediate Command interrupt response</td>
<td>2.3 µs</td>
</tr>
<tr>
<td>Internal CPU reconfigures FIFO SFRs</td>
<td>0.75 µs</td>
</tr>
<tr>
<td>Channel Boundary Pointer SFR</td>
<td>0.75 µs</td>
</tr>
<tr>
<td>Input Threshold SFR</td>
<td>0.75 µs</td>
</tr>
<tr>
<td>Output Threshold SFR</td>
<td>0.75 µs</td>
</tr>
<tr>
<td>Internal CPU resets Slave Control (SLCON) Freeze Mode bit = 1, Normal Mode, and automatically resets Host Status FIFO DMA Freeze Mode bit</td>
<td>2.3 µs</td>
</tr>
<tr>
<td>Internal CPU writes Immediate Command Out</td>
<td>0.75 µs</td>
</tr>
<tr>
<td>Host Immediate Command interrupt service</td>
<td>2.3 µs</td>
</tr>
<tr>
<td>Total Minimum Time to Reconfigure FIFO</td>
<td>112.33 µs</td>
</tr>
</tbody>
</table>

Figure 9. Sequence of Events to Invoke FIFO DMA Freeze Mode and Timings
Table 10. Slave Bus Interface Status During FIFO DMA Freezer Mode

<table>
<thead>
<tr>
<th>Interface Pins;</th>
<th>Operation In Normal Mode</th>
<th>Status In Freeze Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>DACK CS A2 A1 A0 READ WRITE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 0 0 1</td>
<td>Read Host Status SFR</td>
<td>Operational</td>
</tr>
<tr>
<td>1 0 0 1 1 0 1</td>
<td>Read Host Control SFR</td>
<td>Operational</td>
</tr>
<tr>
<td>1 0 0 1 1 1 0</td>
<td>Write Host Control SFR</td>
<td>Disabled</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 1</td>
<td>Data or DMA data from Output Channel</td>
<td>Disabled</td>
</tr>
<tr>
<td>1 0 0 0 0 0 1 0</td>
<td>Data or DMA data to Input Channel</td>
<td>Disabled</td>
</tr>
<tr>
<td>1 0 0 0 1 0 1</td>
<td>Data Stream Command from Output Channel</td>
<td>Disabled</td>
</tr>
<tr>
<td>1 0 0 0 1 1 0</td>
<td>Data Stream Command to Input Channel</td>
<td>Disabled</td>
</tr>
<tr>
<td>1 0 1 0 0 0 1</td>
<td>Read Immediate Command Out from Output Channel</td>
<td>Disabled</td>
</tr>
<tr>
<td>1 0 1 0 0 1 0</td>
<td>Write Immediate Command In to Input Channel</td>
<td>Disabled</td>
</tr>
<tr>
<td>0 X X X X X 0 1</td>
<td>DMA Data from Output Channel</td>
<td>Disabled</td>
</tr>
<tr>
<td>0 X X X X 1 0</td>
<td>DMA Data to Input Channel</td>
<td>Disabled</td>
</tr>
</tbody>
</table>

NOTE: X = don't care

Table 11. FIFO SFR's Characteristics During FIFO DMA Freeze Mode

<table>
<thead>
<tr>
<th>Label</th>
<th>Name</th>
<th>Normal Operation (SST5 = 1)</th>
<th>Freeze Mode Operation (SST5 = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCON</td>
<td>Host Control</td>
<td>Not Accessible</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>HSTAT</td>
<td>Host Status</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>SCON</td>
<td>Slave Control</td>
<td>Read &amp; Write</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>SSTAT</td>
<td>Slave Status</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>IEP</td>
<td>Interrupt Enable &amp; Priority</td>
<td>Read &amp; Write</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>MODE</td>
<td>Mode Register</td>
<td>Read &amp; Write</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>IWPR</td>
<td>Input FIFO Write Pointer</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>IRPR</td>
<td>Input FIFO Read Pointer</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>OWPR</td>
<td>Output FIFO Write Pointer</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>ORPR</td>
<td>Output FIFO Read Pointer</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>CBP</td>
<td>Channel Boundary Pointer</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>IMIN</td>
<td>Immediate Command In</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>IMONT</td>
<td>Immediate Command Out</td>
<td>Read &amp; Write</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>FIN</td>
<td>FIFO IN</td>
<td>Read Only</td>
<td>Read Only</td>
</tr>
<tr>
<td>CIN</td>
<td>COMMAND IN</td>
<td>Read Only</td>
<td>Read Only</td>
</tr>
<tr>
<td>FOUT</td>
<td>FIFO OUT</td>
<td>Read &amp; Write</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>COUT</td>
<td>COMMAND OUT</td>
<td>Read &amp; Write</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>ITHR</td>
<td>Input FIFO Threshold</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
<tr>
<td>OTHR</td>
<td>Other FIFO Threshold</td>
<td>Read Only</td>
<td>Read &amp; Write</td>
</tr>
</tbody>
</table>
Flexibility in Frame Size with the 8044

PARVIZ KHODADADI
APPLICATIONS ENGINEER
1.0 INTRODUCTION

The 8044 is a serial communication microcontroller known as the RUPI (Remote Universal Peripheral Interface). It merges the popular 8051 8-bit microcontroller with an intelligent, high performance HDLC/SDLC serial communication controller called the Serial Interface Unit (SIU). The chip provides all features of the microcontroller and supports the Synchronous Data Link Control (SDLC) communications protocol.

There are two methods of operation relating to frame size:
1) Normal operation (limited frame size)
2) Expanded operation (unlimited frame size)

In Normal operation the internal 192 byte RAM is used as the receive and transmit buffer. In this operation, the chip supports data rates up to 2.4 Mbps externally clocked and 375 Kbps self-clocked. For frame sizes greater than 192 bytes, Expanded operation is required. In Expanded operation the external RAM, in conjunction with the internal RAM, is used as the transmit and receive buffer. In this operation, the chip supports data rates up to 500 Kbps externally clocked and 375 Kbps self-clocked. In both cases, the SIU handles many of the data link functions in hardware, and the chip can be configured in either Auto or Flexible mode.

The discussion that follows describes the operation of the chip and the behavior of the serial interface unit. Both Normal and Expanded operations will be further explained with extra emphasis on Expanded operation and its supporting software. Two examples of SDLC communication systems will also be covered, where the chip is used in Expanded operation. The discussion assumes that the reader is familiar with the 8044 data sheet and the SDLC communications protocol.

1.1 Normal Operation

In Normal operation the on-chip CPU and the SIU operate in parallel. The SIU handles the serial communication task while the CPU processes the contents of the on-chip transmit and receiver buffer, services interrupt routines, or performs the local real time processing tasks.

The 192 bytes of on-chip RAM serves as the interface buffer between the CPU and the SIU, used by both as a receive and transmit buffer. Some of the internal RAM space is used as general purpose registers (e.g. R0–R7). The remaining bytes may be divided into at least two sections: one section for the transmit buffer and the other section for the receive buffer. In some applications, the 192 byte internal RAM size imposes a limitation on the size of the information field of each frame and, consequently, achieves less than optimal information throughput.

Figure 1 illustrates the flow of data when internal RAM is used as the receive and transmit buffer. The on-chip CPU allocates a receive buffer in the internal RAM and enables the SIU. A receiving SDLC frame is processed by the SIU and the information bytes of the frame, if any, are stored in the internal RAM. Then, the SIU informs the CPU of the received bytes (Serial Channel interrupt). For transmission, the CPU loads the transmitting bytes into the internal RAM and enables the SIU. The SIU transmits the information bytes in SDLC format.
1.2 Expanded Operation

In Expanded operation the on-chip CPU monitors the state of the SIU, and moves data from/to external buffer to/from the internal RAM and registers while reception/transmission is taking place. If the CPU must service an interrupt during transmission or reception of a frame or transmit from internal RAM, the chip can shift to Normal operation.

There is a special function register called SIUST, the contents of which dictate the operation of the SIU. Also, at data rates lower than 2.4 Mbps, one section of the SIU, in fixed intervals during transmission and reception, is in the "standby" mode and performs no function. The above two characteristics make it possible to program the CPU to move data to/from external RAM and to force the SIU to repeat or skip some desired hardware tasks while transmission or reception is taking place. With these modifications, external RAM can be utilized as a transmit and receive buffer instead of the internal RAM.

Figure 2 graphically shows the flow of data when external RAM is used. For reception, the receiving bytes are loaded into the Receive Control Byte (RCB) register. Then, the data in RCB is moved to external RAM and the SIU is forced to load the next byte into the RCB register - the chip believes it is receiving a control byte continuously. For transmission, Information bytes (I-bytes) are loaded into a location in the internal RAM and the chip is forced to transmit the contents of this location repeatedly.

Discussion of expanded operation is continued in sections 4 and 5. First, however, sections 2 and 3 describe features of the 8044 which are necessary to further explain expanded operation.

2.0 THE SERIAL INTERFACE UNIT

2.1 Hardware Description

The Serial Interface Unit (SIU) of the RUPI, shown in Figure 3, is divided functionally into a Bit Processor (BIP) and a Byte Processor (BYP), each sharing some common timing and control logic. The bit processor is the interface between the SIU bus and the serial port pins. It performs all functions necessary to transmit/receive a byte of data to/from the serial data line (shifting, NRZI coding, zero insertion/deletion, etc.). The byte processor manipulates bytes of data to perform message formatting, transmitting, and receiving functions. For example, moving bytes from/to the special function registers to/from the bit processor.

The byte processor is controlled by a Finite-State Machine (FSM). For every receiving/transmitting byte, the byte processor executes one state. It then jumps to the next state or repeats the same state. These states will be explained in section 3. The status of the FSM is kept in an 8-bit register called SIUST (SIU State Counter). This register is used to manipulate the behavior of the byte processor.

As the name implies, the bit processor processes data one bit at a time. The speed of the bit processor is a function of the serial channel data rate. When one byte of data is processed by the bit processor, a byte bounda-
FLEXIBILITY IN FRAME SIZE WITH THE 8044

ry is reached. Each time a byte boundary is detected in the serial data stream, a burst of clock cycles (16 CPU states) is generated for the byte processor to execute one state of the state machine. When all the procedures in the state are executed, a wait signal is asserted to terminate the burst, and the byte processor waits for the next byte boundary (standby mode). The lower the data rate, the longer the byte processor will stay in the standby mode.

2.2 Reception of Frames

Incoming data is NRZI decoded by the on-chip decoder. It is then passed through the zero insertion/deletion (ZID) circuitry. The ZID not only performs zero insertion/deletion, but also detects flags and Go Aheads (GA) in the data stream. The data bits are then loaded into the shift register (SR) which performs serial to parallel conversion. When 8 bits of data are collected in the shift register, the bit processor triggers the byte processor to process the byte, and it proceeds to load the next block of data into the shift register. The serial data is also shifted, through SR, to a 16-bit register called "FCS GEN/CHK" for CRC checking. The byte processor takes the received address and control bytes from the SR shift register and moves them to the appropriate registers. If the contents of the shift register is expected to be an information byte, the byte processor moves them through a 3-byte FIFO to the internal RAM at a starting location addressed by the contents of the Receive Buffer Start (RBS) register.

2.3 Transmission of Frames

In the transmit mode, the byte processor relinquishes a byte to the bit processor by moving it to a register called RB (RAM buffer). The bit processor converts the data to serial form through the shift register, performs zero bit insertion, NRZI encoding, and sends the data to the serial port for transmission. Finally, the contents of the FCS GEN/CHK and the closing flag are routed to the serial port for transmission.

Figure 3. SIU Block Diagram


3.0 TRANSMIT AND RECEIVE STATES

The simplified receive and transmit state diagrams are shown in Figures 4 and 5, respectively. The numbers on the left of each state represent the contents of the SIUST register when the byte processor is in the standby mode, and the instructions on the right of each state represent the “state procedures” of that state. When the byte processor executes these procedures the least three significant bits of the SIUST register are being incremented while the other bits remain unchanged. The byte processor will jump from one state to another without going into the standby mode when a conditional jump procedure executed by the byte processor is true.

3.1 Receive State Sequence

When an opening flag (7EH) is detected by the bit processor, the byte processor is triggered to execute the procedures of the FLAG state. In the FLAG state, the byte processor loads the contents of the RBS register into the Special RAM (SRAR) register. SRAR is the pointer to the internal RAM. The byte processor decrements the contents of the Receive Buffer Length (RBL) register and loads them into the DMA Count (DCNT) register. The FCS GEN/CHK circuit is turned on to monitor the serial data stream for Frame Check Sequence functions as per SDLC specifications.

Assuming there is an address field in the frame, contents of the SIUST register will then be changed to 08H, causing the byte processor to jump to the ADDRESS state and wait (standby) for the next byte boundary. As soon as the bit processor moves the address byte into the SR shift register, a byte boundary is achieved and the byte processor is triggered to execute the procedures in the ADDRESS state.

In the ADDRESS state the received station address is compared to the contents of the STAD register. If there is no match, or the address is not the broadcast address (FFH), reception will be aborted (SIUST = 01H). Otherwise, the byte processor jumps to the CONTROL state (SIUST = 10H) and goes into standby mode.

The byte processor jumps to the CONTROL state if there exists a control field in the receiving frame. In this state the control byte is moved to the RCB register by the byte processor. Note that the only action taken in this state is that a received byte, processed by the bit processor, is moved to RCB. There is no other hardware task performed, and DCNT and SRAR are not affected in this state.

The next two states, PUSH-1 and PUSH-2, will be executed if Frame check sequence (NFCS = 0) option is selected. In these two states the first and second bytes of the information field are pushed into the 3-byte FIFO (FIFO0, FIFO1, FIFO2) and the Receive Field Length register (RFL) is set to zero. The 3-byte FIFO is used as a pipeline to move received bytes into the internal RAM. The FIFO prevents transfer of CRC bytes and the closing flag to the receive buffer (i.e., when the ending flag is received, the contents of FIFO are FLAG, FCS1, and FCS0.) The three byte FIFO is collapsed to one byte in No FCS mode.

In the DMA-LOOP state the byte processor pushes a byte from SR to FIFO0, moves the contents of FIFO2 to the internal RAM addressed by the contents of SRAR, increments the SRAR and RFL registers, and decrements the DCNT register. If more information bytes are expected, the byte processor repeats this state on the next byte boundaries until DMA Buffer End occurs. The DMA Buffer End occurs if SRAR reaches 0BFH (192 decimal), DCNT reaches zero, or the RBP bit of the STS register is set.

The BOV-LOOP state, the last state, is executed if there is a buffer overrun. Buffer overrun occurs when the number of information bytes received is larger than the length of the receive buffer (RFL > RBL). This state is executed until the closing flag is received.

At the end of reception, if the FCS option is used, the closing flag and the FCS bytes will remain in the 3-byte FIFO. The contents of the RCB option are used to update the NSNR (Receive/Send Count) register. The SIU updates the STS register and sets the serial interrupt.

3.2 Transmit State Sequence

Setting the RTS bit puts the SIU in the transmit mode. When the CTS pin goes active, the byte processor goes into START-XMIT state. In this state the opening flag is moved into the RAM Buffer (RB) register. The byte processor jumps to the next state and goes into the standby mode.

If the Pre-Frame Sync (PFS) option is selected, the PFS1 and PFS2 states will be executed to transmit the two Pre-Frame Sync bytes (00H or 55H). In these two states the contents of the Pre-Frame Sync generator are sent to the serial port while the Zero Insertion Circuit (ZID) is turned off. ZID is turned back on automatically on the next byte boundary.

If the PFS option is not chosen, the byte processor jumps to the FLAG state. In this state, the byte processor moves the contents of TBS into the SRAR register, decrements TBL and moves the contents into the DCNT register. The byte processor turns off the ZID and turns on FCS GEN/CHK. The contents of FCS GEN/CHK are not transmitted unless the NFCS bit is
set. If a frame with the address field is chosen, it moves the contents of the STAD register into the RB register for transmission. At the same time, the opening flag is being transmitted by the bit processor.

In the ADDRESS (SIUST = A0H) and CONTROL (SIUST = A8H) states, TCB and the first information byte are loaded into, the RB register for transmission, respectively. Note that in the CONTROL state, none of the registers (e.g. DCNT, SRAR) are incremented, and ZID and FCS GEN/CHK are not turned on or off.

The procedures in the DMA-LOOP state are similar to the procedures of the DMA-LOOP in the receive state diagram. The SRAR register pointer to the internal RAM is incremented, and the DCNT register is decremented. The contents of DCNT reach zero when all the information bytes from the transmit buffer are transmitted. A byte from RAM is moved to the RB register for transmission. This state is executed on the following byte boundaries until all the information bytes are transmitted.

The FCS1 and the FCS2 states are executed to transmit the Frame Check Sequence bytes generated by the FCS generator, and the END-FLAG state is executed to transmit the closing flag.

The XMIT-ACTION and the ABORT-ACTION states are executed by the byte processor to synchronize the SIU with the CPU clock. The XMIT-ACTION or the ABORT-ACTION state is repeated until the byte processor status is updated. At the end, the STS and the TMOD registers are updated.

The two ABORT-SEQUENCE states (SIUST = E0H and SIUST = E8H) are executed only if transmission is aborted by the CPU (RTS or TBF bit of the STS register is cleared) or by the serial data link (CTS signal goes inactive or shut-off occurs in loop mode.)
Figure 5. Transmit State Diagram
4.0 TRANSMISSION/RECEPTION OF LONG FRAMES (EXPANDED OPERATION)

In this application note, a frame whose information field is more than 192 bytes (size of on-chip RAM) is referred to as a long frame. The 8044 can access up to 64000 bytes of external RAM. Therefore, a long frame can have up to 64000 information bytes.

4.1 Description

During transmission or reception of a frame, while the bit processor is processing a byte, the byte processor, after 16 CPU states, is in the standby mode, and the internal registers and the internal bus are not used. The period between each byte boundary, when the byte processor is in the standby mode, can be used to move data from external RAM to one of the byte processor registers for transmission and vice versa for reception. The contents of the SIUST register, which dictate the state of the byte processor, can be monitored to recognize the beginning of each SDLC field and the consecutive byte boundaries.

By writing into the SIUST register, the byte processor can be forced to repeat or skip a specific state. As an example, the SIU can be forced to repeatedly put the received bytes into the RCB register. This is accomplished by writing E7H into the SIUST register when the byte processor goes into the standby mode. The byte processor, therefore, executes the CONTROL state at the next byte boundary.

For transmission, the byte processor is put in the transmit mode. When transmission of a frame is initiated, the user program calls a subroutine in which the state of the byte processor is monitored by checking the contents of the SIUST register. When the byte processor reaches a desired state and goes into standby, the CPU loads the first byte of the internal RAM buffer with data and moves the byte processor to the CONTROL state. The routine is repeated for every byte. At the end, the program returns from the subroutine, and the SIU finishes its task (see application examples).

For reception, a software routine is executed to move data to external RAM and to force the SIU to repeat the CONTROL state. The CONTROL state is repeated because, as shown in the receive state diagram, the only action taken by the byte processor, in the CONTROL state, is to move the contents of SR to the RCB register. None of the registers (e.g. SRAR and DCNT) are incremented. A similar comment justifies the use of the CONTROL state for transmission. In the transmit CONTROL state, contents of a location in the on-chip RAM addressed by TBS is moved to RB for transmission.

4.2 SIU Registers

To write into the SIUST register, the data must be complemented. For example, if you intend to write 18H into the SIUST register, you should write E7H to the register. The data read from SIUST is, however, true data (i.e. 18H).

Read and write accesses to the SIUST, STAD, DCNT, RCB, RBL, RFL, TCB, TBL, TBS, and the 3-byte FIFO registers are done on even and odd phases, respectively. Therefore, there is no bus contention when the CPU is monitoring the registers (e.g. SIUST), and SIU is simultaneously writing into them.

There is no need to change or reset the contents of any SIU register while transmitting or receiving long frames, unless the byte processor is forced to repeat a state in which the contents of these registers are modified. Note that the SRAR register can not be accessed by the CPU; therefore, avoid repeating the DMA-LOOP states. If SRAR increments to 192, the SIU will be interrupted and communication will be aborted.

4.3 Other Possibilities

The internal RAM, in conjunction with an external buffer (RAM or FIFOs), can be used as a transmit and receive buffer. In other words, Expanded and Normal operation can be used together. For example, if a frame with 300 Information bytes is received and only 255 of them are moved to an external buffer, the remaining bytes (45 bytes) will be loaded into the internal RAM by the SIU (assuming RBL is set to 45 or more). The contents of RFL indicate the number of bytes stored in the internal RAM. For transmission, the contents of the external buffer can be transmitted followed by the contents of the internal buffer.

If the internal RAM is not used, contents of the RBL register can be 0 and contents of the TBL register must be set to 1. The contents of the TBS register can be any location in the internal RAM.

The transmission and reception procedures for long frames with no FCS are similar to those with FCS. The exception is the contents of the SIUST register should be compared with different values since the two FCS states of the transmit and receive flow charts are skipped by the byte processor.

If a frame format with no control byte is chosen, a location in the RAM addressed by TBS should be used for transmission as with control byte format. The FIFO can be used for reception. The STAD register can be used for transmission if no zero insertion is required.
If the RUPI is used in Auto mode (see Section 5), it will still respond to RR, RNR, REJ, and Unnumbered Poll (UP) SDLC commands with RR or RNR automatically, without using any transmit routine. For example, if the on-chip CPU is busy performing some real time operations, the SIU can transmit an information frame from the internal buffer or transmit a supervisory frame without the help of CPU (Normal operation).

Maximum data rate using this feature is limited primarily by the number of instructions needed to be executed during the standby mode.

Transmission or reception of a frame can be timed out so that the CPU will not hang up in the transmit or receive procedures if a frame is aborted. Or, if the data rate allows enough time (standby time is long enough), the CPU can monitor the SIUST register for idle mode (SIUST = 01H).

It is also possible to transmit multiple opening or closing flags by forcing the byte processor to repeat the END-FLAG state.

4.4 Maximum Data Rate in Expanded Operation

Assuming there is no zero-insertion/deletion, the bit processor requires eight serial clock periods to process one block of data. The byte processor, running on the CPU clock, processes one byte of data in 16 CPU states (one state of the state diagrams). Each CPU state is two oscillator periods. At an oscillator frequency of 12 MHz, the CPU clock is 6 MHz, and 16 CPU states is 2.7 μs. At a 3 Mbit rate with no zero-insertion/deletion, there is exactly enough time to execute one state per byte (16 states at 6 MHz = 8 bits at 3M baud). In other words, the standby time is zero.

Figure 6 demonstrates portions of the timing relationship between the byte processor and the bit processor. In each state, the actions taken by the processors, plus the contents of the SIUST register, are shown. When the byte processor is running, the contents of SIUST are unknown. However, when it is in the standby mode, its contents are determinable.

The maximum data rate for transmitting and receiving long frames depends on the number of instructions needed to be executed during standby, and is propor-

---

**Figure 6. Portions of the BIP/BYP Timing Relationship**

<table>
<thead>
<tr>
<th>STATE:</th>
<th>ADDRESS</th>
<th>CONTROL</th>
<th>PUSH-1</th>
<th>PUSH-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIP:</td>
<td>CRTL BYTE → SR</td>
<td>1st l=BYTE → SR</td>
<td>2nd l=BYTE → SR</td>
<td></td>
</tr>
<tr>
<td>BYP:</td>
<td>SR → TMP</td>
<td>STANDBY</td>
<td>SR → RCB</td>
<td>STANDBY</td>
</tr>
</tbody>
</table>

6a. Reception

<table>
<thead>
<tr>
<th>STATE:</th>
<th>FLAG</th>
<th>ADDRESS</th>
<th>CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIP:</td>
<td>XMITTING FLAG</td>
<td>XMITTING ADRS BYTE</td>
<td>XMITTING CRTL BYTE</td>
</tr>
<tr>
<td>BYP:</td>
<td>STAD → RB</td>
<td>STANDBY</td>
<td>TCB → RB</td>
</tr>
<tr>
<td>SIUST:</td>
<td>?</td>
<td>00H</td>
<td>?</td>
</tr>
</tbody>
</table>

6b. Transmission

---
FLEXIBILITY IN FRAME SIZE WITH THE 8044

5.0 MODES OF OPERATION

The 8044 has two modes: Flexible mode and Auto mode. In Auto mode, the chip responds to many SDLC commands and keeps track of frame sequence numbering automatically without on-chip CPU intervention. In Flexible mode, communication tasks are under control of the on-chip CPU.

5.1 Flexible Mode

For transmission, the CPU allocates space for transmit buffer by storing values for the starting location and size of the transmit buffer in the TBS and the TBL registers. It loads the buffer with data, sets the TBF and the RTS bits in the STS register, and proceeds to perform other tasks. The SIU activates the RTS line. When the CTS signal goes active, the SIU transmits the frame. At the end of transmission, the SIU clears the RTS bit and interrupts the CPU (SI set).

For reception, the CPU allocates space for receive buffer by loading the beginning address and length of the receive buffer into the RBS and RBL registers, sets the RBE bit, and proceeds to perform other tasks. The SIU, upon detection of an opening flag, checks the next received byte. If it matches the station address, it will load the received control byte into RCB, and received information bytes into the receive buffer. At the end of reception, if the Frame Check Sequence (FCS) is correct, the SIU clears RBE and interrupts the CPU.

5.2 Auto Mode

In the Auto mode, the 8044 can only be a secondary station operating in the SDLC "Normal Response Mode". The 8044 in Auto mode does not transmit messages unless it is polled by the primary.

For transmission of an information frame, the CPU allocates space for the transmit buffer, loads the buffer with data, and sets the TBF bit. The SIU will transmit the frame when it receives a valid poll-frame. A frame whose poll bit of the control byte is set, is a poll-frame. The poll bit causes the RTS bit to be set. If TBF were not set, the SIU would respond with Receive Not Ready (RNR) SDLC command if RBP = 1, or with Receive Ready (RR) SDLC command if RBP = 0. After transmission RTS is cleared, and the CPU is not interrupted.

For reception, the procedure is the same as that of Flexible mode. In addition, the SIU sets the RTS bit if the received frame is a poll-frame (causing an automatic response) and increments the NS and NR counts accordingly.

6.0 APPLICATION EXAMPLES

Two application examples are given to provide additional details about the procedures used to transmit and receive long frames. In the first application example, procedures to construct receive and transmit software routines for the point-to-point frame format are described. The point-to-point frame has the information field and the FCS field enclosed between two flags (see Figure 7). In the second example software code is generated for reception and transmission of the standard SDLC frame. The SDLC frame has the pattern: flag, address, control, information, FCS, flag.

The first example focuses on the construction of transmit and receive code which allow the chip to transmit and receive long frames. The second example shows how to make more use of the 8044 features, such as the on-chip phase-locked loop for clock recovery and automatic responses in the Auto mode to demonstrate the capability of the 8044 to achieve high throughput when Expanded operation is used.

6.1 Point-to-Point Application Example

A point-to-point communication system was developed to receive and transmit long frames. The system consists of one primary and one secondary station. Although multiple secondary stations can be used in this
system, one secondary is chosen to simplify the primary station's software and focus on the long frame software code. Both the primary and the secondary stations are in Flexible mode and the external clock option is used for the serial channel. The maximum data rate is 500 Kbps. The FCS bytes are generated and checked automatically by both stations.

6.1.1 POLLING SEQUENCE

The polling sequence, shown in Figure 8, takes place continuously between the primary and the secondary stations. The primary transmits a frame with one information byte to the secondary. The information byte is used by the secondary as an address byte. The secondary checks the received byte, and if the address matches, the secondary responds with a long frame. In this example, the information field of the frame is chosen to be 255 bytes long. Since there is only one secondary station, the address always matches. Upon successful reception of the long frame, the primary transmits another frame to the secondary station.

6.1.2 HARDWARE

The schematic of the secondary station is given in Figure 9. The circuit of the primary station is identical to the secondary station with the exception of pin 11 (DATA) being connected to pin 14 (T0). In the primary station, the 8044 is interrupted when activity is detected on the communication line by the on-chip timer (in counter mode). This is explained more later. The serial clock to both stations is supplied by a pulse generator. The output of the pulse generator (not shown in the diagram) is connected to pin 15 of the 8044s. Since the two stations are located near each other (less than 4 feet), line drivers are not used.

The central processor of each station is the 8044. The data link program is stored in a 2Kx8 EPROM (2732A), and a 2Kx8 static RAM (AM9128) is used as the external transmit and receive buffer. The RTS pin is connected to the CTS pin. For simplicity, the stations are assumed to be in the SDLC Normal Respond Mode after Hardware reset.

6.1.3 PRIMARY STATION SOFTWARE

The assembly code for the primary station software is listed in Appendix A. The primary software consists of the main routine, the SIU interrupt routine, and the receive interrupt routine. The receive interrupt routine is executed when a long frame is being received.

In the flow charts that follow, all actions taken by the SIU appear in squares, and actions taken by the on-chip CPU appear in spheres.
Figure 9. Secondary Station Hardware

Flexibility in Frame Size with the 8044

Intel
Main Routine

First, the chip is initialized (see Figure 10). It is put in Flexible mode, externally clocked, and “Flag-Information Field-FCS-Flag” frame format. Pre-Frame Sync option (PFS = 1) and automatic Frame Check Sequence generation/detection (NFCS = 0) are selected. The on-chip transmit buffer starts at location 20H and the transmit buffer length is set to 1. This one byte buffer contains the address of the secondary station. There is no on-chip receive buffer since the long frame being received is moved to the external buffer. The RTS, TBF, and RBE bits are set simultaneously. Setting the RTS and TBF bits causes the SIU to transmit the contents of the transmit buffer.

SIU Interrupt Routine

After transmission of the frame, the SIU interrupts the on-chip CPU (SI is set). In the SIU interrupt service routine, counter 0 is initialized and turned on (see Figure 11). The user program returns to perform other tasks. After reception of the long frame, the SIU interrupt routine is executed again. This time, RTS, TBF, and RBE are set for another round of information exchange between the two stations.

SIU never interrupts while reception or transmission is taking place. The SIU registers are updated and the SI is set (serial interrupt) after the closing flag has been received or transmitted. An SIU interrupt never occurs if the receive interrupt routine or the transmit subroutine is being executed.

Setting the RBE bit of the STS register puts the RUPI in the receive mode. However, the jump to the receive interrupt routine occurs only when a frame appears on the serial port. Incoming frames can be detected using the Pre-Frame Sync. option and one of the CPU timers in counter mode. The counter external pin (T0) is connected to the data line (pin 11 is tied to pin 14). Setting the PFS (Pre-Frame Sync.) bit will guarantee 16 transitions before the opening flag of a frame.
The counter registers are initialized such that the counter interrupt occurs before the opening flag of a frame. When the PFS transitions appear on the data line, the counter overflows and interrupts the CPU. The CPU program jumps to the timer interrupt service routine and executes the receive routine. In the receive routine, the received frame is processed, and the information bytes are moved to the external RAM. Note that the maximum count rate of the 8051 counter is \( \frac{1}{24} \) of the oscillator frequency. At 12 MHz, the data rate is limited to 500 Kbps.

Another method to detect a frame on the data line and cause an interrupt is to use an external "Flag-Detect" circuit to interrupt the CPU. The "Flag Detect" circuit can be an 8-bit shift register plus some TTL chips. If this option is used, the RUPI must operate in externally clocked mode because the clock is needed to shift the incoming data into the shift register. With this option, the maximum data rate is not limited by the maximum count rate of the 8051 counter.

### Receive Interrupt Routine

In Normal operation, the byte processor executes the procedures of the FLAG state, jumps to the CONTROL state without going into the standby mode, and executes 10–2 procedure of the state (see Figure 4). It then jumps to the PUSH-1 state and goes into the standby mode. At the following byte boundaries, the byte processor executes the PUSH-1, PUSH-2, and DMA-LOOP states, respectively. The receive interrupt routine as shown in the flow chart of Figure 12 and described below forces the byte processor to repeatedly execute the CONTROL state before the PUSH-1 state is executed. The following is the step by step procedure to receive long frames:

1) Turn off the CPU counter and save all the important registers. Jump to the receive interrupt routine, execution of the instructions to save registers, and initialization of the receive buffer pointer take place while the Pre-Frame Sync bytes and the opening flag are being received. This is about three data byte periods (48 CPU cycles at 500 Kbps).

2) Monitor the SIUST register for standby in the PUSH-1 state (SIUST = 18H). When the SIUST contents are 18H, the byte processor is waiting for the first information byte. The bit processor has already recognized the flag and is processing the first information byte.

3) In the standby mode, move the byte processor into the CONTROL state by writing "EFH" (complement of 10H) into the SIUST register. When the next byte boundary occurs, the bit processor has processed and moved a byte of data into the SR register. The byte processor moves the contents of SR into the RCB register, jumps to the PUSH-1 state (SIUST = 18H), and waits.

4) Monitor the SIUST register for standby in the PUSH-1 state. When the contents of SIUST becomes 18H, the contents of RCB are the first information byte of the information field.

5) While the byte processor is in the standby mode, move the contents of RCB to an external RAM or an I/O port.

6) Check for the end of the information field. The end can be detected by knowing the number of bytes transmitted, or by having a unique character at the end of information field. The length of the information field can be loaded into the first byte(s) received. The receive routine can load this byte into the loop counter.

7) If the byte received is not the last information byte, move the byte processor back to standby in the CONTROL state and repeat steps 4 through 6. Otherwise, return from the interrupt routine.

Upon returning from the receive interrupt routine, the byte processor automatically executes the PUSH-1, PUSH-2, and DMA-LOOP before it stops. This causes the remaining information bytes (if any) to be stored in the internal RAM at the starting location specified by the contents of RBS register. At the end of the cycle, the closing flag and the CRC bytes are left in the FIFO. The RFL register will be incremented by the number of bytes stored in the internal RAM. Then, the STS and NSNR registers are updated, and an appropriate response is generated by the SIU.

The software to perform the above task is given in Table 1. In this example, the number of instruction cycles executed during standby is 12 cycles.
6.1.4 SECONDARY STATION SOFTWARE

The assembly code for the secondary station software is given in Appendix A. The secondary station contains the transmit subroutine which is called for transmission of long frames.

**Main Routine**

As shown in the secondary station flow chart (Figure 13), the external transmit buffer (external RAM) is loaded with the information data (FFH, FEH, FDH, ...) at starting location 200H. The internal transmit buffer (on chip RAM) starts at location 20H (TBS = 20H), and the transmit buffer length (TBL) is set to 1. The on-chip CPU, in the transmit subroutine, moves the information bytes from the external RAM to this one byte buffer for transmission. The receive buffer starts at location 10H and the receiver buffer length is 1. This buffer is used to buffer the frame transmitted by the primary. The received byte is used as an address byte.

The Secondary is configured like the Primary station. It is put in Flexible mode, externally clocked, Point-to-point frame format. The PFS bit is set to transmit two bytes before the first flag of a frame. The RBE bit is set to put the chip in receive mode. Upon reception of a valid frame, the SIU loads the received information byte into the on-chip receive buffer and interrupts the CPU.

**SIU Interrupt Routine**

In the serial interrupt routine, the RBE bit is checked (see Figure 14). Since RBE is clear, a frame has been received. The received Information byte is compared with the contents of the Station Address (STAD) register.
If they match, the secondary will call the transmit subroutine to transmit the long frame. Upon returning from the transmit subroutine, the RBE bit is set, and program returns from the SIU interrupt. After transmission of the closing flag, SIU interrupt occurs again. In the interrupt routine, the RBE is checked. Since the RBE is set, the program returns from the SIU interrupt routine and waits until another long frame is received.

If the secondary were in Auto mode, the chip must be ready to execute the transmit routine upon reception of a poll-frame; otherwise, the chip automatically transmits the contents of the internal transmit buffer if the TBF bit is set, or transmits a supervisory command (RR or RNR) if TBF is clear.

**Transmit Subroutine**

In Normal operation the byte processor executes the START-TRANSMIT state and jumps to the PFS1 state. While the bit processor is transmitting some unwanted bits, the byte processor executes the PFS1 state and jumps to the standby mode in the PFS2 state.

While the bit processor is transmitting the first Pre-Frame Sync byte, the byte processor executes the PFS2 state and jumps to the standby mode in the FLAG state. The FLAG state is executed when the bit processor begins to transmit the second Pre-Frame Sync byte. When the flag is being transmitted, the byte processor executes the 98-1, 98-2, 98-3, and 98-4 procedures of the FLAG state, and jumps to execute the A8-1 procedure of the CONTROL state. When the opening flag is transmitted, the contents of RB are the first information byte. (See transmit State diagram.)

In the transmit subroutine (see Figure 15), the byte processor is forced to repeat the CONTROL state before the DMA-LOOP state. In the CONTROL state, the contents of a RAM location addressed by the TBS register are moved to the RB register. The following is the step by step procedure to transmit long frames:

1) Put the chip in transmit mode by setting the RTS and TBF bits.
2) Move an information byte from external RAM to a location in the internal RAM addressed by the contents of TBS.
3) Monitor the SIUST register for the standby mode in the DMA-LOOP state (SIUST = B0H). When SIUST is B0H, the opening flag has been transmitted, and the first information byte is being transmitted by the bit processor.

4) If there are more information bytes, move the byte processor back to the CONTROL state, and repeat steps 2 through 4. Otherwise, continue.

5) Move byte processor to the Standby mode in the CONTROL state (SIUST = A8H) and return from the subroutine.

The byte processor automatically executes the remaining states to send the FCS bytes and the closing flag. After the completion of transmission, SIU updates the STS and NSNR registers and interrupts the CPU.

If the contents of the TBL register were more than 1, the SIU transmits (TBL) – 1 additional bytes from the internal RAM at starting address (TBS) + 1 because it executes the DMA-LOOP state (TBL) – 1 additional times. The byte processor should not be programmed to skip the DMA-LOOP state, because the transmission of FCS bytes is enabled in this state.

The maximum baud rate that can be used with these codes is calculated by adding the number of instruction cycles executed, during the standby mode, between each byte boundaries (see Table 2).

Using Equation 1, the maximum data rate, based on the transmit software, is 509 Kbps; However, the maximum count rate of the counter limits the data rate to 500 Kbps.

### Table 2. Codes for Long Frame Transmission

<table>
<thead>
<tr>
<th>Transmit Codes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAN: MOV D PTR, #200H</td>
<td>13 Cycles</td>
</tr>
<tr>
<td>MOV R5, #0FFH</td>
<td></td>
</tr>
<tr>
<td>SETB TBF</td>
<td></td>
</tr>
<tr>
<td>SETB RTS</td>
<td></td>
</tr>
<tr>
<td>LOOP: MOVX A, @DPTR</td>
<td></td>
</tr>
<tr>
<td>MOV @R1, A</td>
<td></td>
</tr>
<tr>
<td>MOV A, #0BOH</td>
<td></td>
</tr>
<tr>
<td>WAIT1: CJNE A, SIUST, WAIT1</td>
<td>2</td>
</tr>
<tr>
<td>INC D PTR</td>
<td>2</td>
</tr>
<tr>
<td>MOVX A, @DPTR</td>
<td>2</td>
</tr>
<tr>
<td>MOV @R1, A</td>
<td>1</td>
</tr>
<tr>
<td>DJNZ R5, NEXTI</td>
<td>2</td>
</tr>
<tr>
<td>MOV SIUST, #57H</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>NEXTI: MOV SIUST, #57H</td>
<td>2</td>
</tr>
<tr>
<td>MOV A, #0BOH</td>
<td>1</td>
</tr>
<tr>
<td>JMP WAIT1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 15. Secondary Station Flow Charts

![Transmit Subroutine](image-url)
6.2 Multidrop Application

Performance of long frame in addition to the features of the 8044 are described using a simple multidrop communication system in which three RUPIs, one as a master and the other two as secondary stations, transmit and receive long frames alternately (see Figure 16). All stations perform automatic zero bit insertion/deletion, NRZI decoding/encoding, Frame Check Sequence (FCS) generation/detection, and on-chip clock recovery at a data rate of 375 Kbps.

The primary and the secondary station’s software code is given in Appendix B. These programs, for simplicity, assume only reception of information and supervisory frames. It is also assumed that the frames are received and transmitted in order. All stations use very similar transmit and receive routines. This code is written for standard SDLC frames (see Figure 7).

6.2.1 POLLING SEQUENCE

The primary station, in Flexible mode, transmits a long frame (for this example, 255 I-bytes), polls one of the secondary stations, and acknowledges a previously received frame simultaneously (see Figure 17). Both secondary stations, in Auto mode, detect the transmitted frame and check its address byte. One of the secondary stations receives the frame, stores the Information bytes in an external RAM buffer, and transmits the same data back to the primary. After reception of the frame, the primary polls and transmits a long frame to the other secondary station which will respond with the same long frame.

6.2.2 HARDWARE

The schematic of the secondary station hardware is shown in Figure 18. The primary station’s hardware is similar to the secondary station’s hardware. The exception is in secondary stations only, where the RTS signal is inverted and tied to the interrupt 0 input pin (INT0). In the primary station, RTS is tied to CTS. At each station, software codes are stored in external EPROM (2732A). Static RAM (2Kx8) is used as external transmit/receive buffer. There is no hardware handshaking done between the stations. The serial clock is extracted from the data line using the on-chip phase locked loop.

![Figure 16. SDLC Multidrop Application Example](image)

![Figure 17. Polling Sequence Between the Primary and Secondary Stations](image)
6.2.3 PRIMARY SOFTWARE

Main Routine

During initialization (see Figure 19), the 8044 is set to Flexible mode, internally clocked at 375 Kbps, and configured to handle standard SDLC frames. The on-chip receive and transmit buffer starting addresses and lengths are selected. The external transmit buffer is chosen from physical location 200H to location 2FFH (255 bytes). The external transmit buffer (external RAM) is loaded with data (FFH, FEH, FDH, FCH, ... DOH). Timer 0 is put in counter mode and set to priority 1. The counter register (TLO) is loaded such that interrupt occurs after 8 transitions on the data line. The Pre-Frame Sync option (setting bit 2 of the SMD register) is selected to guarantee at least 16 transitions before the opening flag of a frame.

The station address register (STAD) is loaded with address of one of the secondary stations. The RTS, TBF, and RBE bits of the STS register are simultaneously set and a call to the transmit routine follows. The transmit routine transmits the contents of the external transmit buffer. At the end of transmission, RTS and TBF are cleared by the SIU, and SIU interrupt occurs. In Flexible mode, SIU interrupt occurs after every transmission or reception of a frame.

SIU Interrupt Routine

In the SIU interrupt service routine (see Figure 20), SI is cleared and the RBE bit is checked. If RBE is set, a long frame has been transmitted. The first time through the SIU interrupt service routine, the RBE test indicates a long frame has been transmitted to one of the secondary stations. Therefore, the Counter is initialized
FLEXIBILITY IN FRAME SIZE WITH THE 8044

and turned on. The program returns from the interrupt routine before a frame appears on the communication channel.

When a frame appears on the communication line, counter interrupt occurs and the receive routine is executed to move the incoming bytes into the external RAM. After reception of the frame and return from the receive routine, SIU interrupt occurs again.

In the SIU interrupt routine, RBE is checked. Since the RBE bit is clear, a frame has been received. Therefore, the appropriate NS and NR counters are incremented and loaded into the TCB register (two pairs of internal RAM bytes keep track of NS and NR counts for the two secondary stations). Transmission of a frame to the next secondary station is enabled by setting the RTS and the TBF bits. The chip is also put in receive mode (RBE set), and a call to transmit routine is made. After transmission, SIU interrupt occurs again, and the process continues.

6.2.4 SECONDARY SOFTWARE

Main Routine

Both secondary stations have identical software (Appendix B). The only differences are the station addresses. Contents of the STAD register are 55H for one station and 44H for the other.

The secondary is configured to transmit an Information frame every time it is polled. The RTS pin is inverted and tied to INT1 pin. External interrupt 1 is enabled and set to interrupt on low to high transition of the RTS signal. This will cause an interrupt (EXI set) after a frame is transmitted. In the interrupt routine the CTS pin is cleared to prevent any automatic response from the secondary. If the CTS pin were not disabled, the secondary station would respond with a supervisory frame (RNR) since the TBF is set to zero by the SIU due to the acknowledge. In the SIU interrupt routine, the CTS pin is cleared after the TBF bit is set. If this option is not used, the primary should acknowledge the previously received frame and poll for the next frame in two separate transmissions.

SIU Interrupt Routine

When a frame is received, counter 0 interrupt occurs and the receive routine is executed (see Figure 22). If the incoming frame is addressed to the station, the information bytes are stored in external RAM; otherwise, the program returns from the receive routine to perform other tasks. At the end of the frame, SIU interrupt occurs. In Auto mode, SIU interrupt occurs whenever an Information frame or a supervisory frame is received. Transmission will not cause an interrupt. In the SIU interrupt service routine, the AM bit of the STS is checked.

If AM bit is set, the interrupt is due to a frame whose address did not match with the address of the station. In this case, NFCS, AM, and the BOV bits are cleared, the RBE bit is set, the counter 0 is initialized and turned on, and program returns from the interrupt routine.

If AM bit is not set, a valid frame has been received and stored in the external RAM. TBF bit is set, CTS pin is activated, counter 0 is disabled and a call to transmit routine is made which transmits the contents of external transmit buffer. This frame also acknowledges the reception of the previously received frame (NS and NR are automatically incremented). Upon return from the transmit routine RBE is set and counter 0 is turned on, thereby putting the chip in the receive mode for another round of data exchange with the primary.

Note that, if the second station is in receive mode, and the counter is enabled and turned on, the CPU will be interrupted each time a frame is on the communication channel. If the frame is not addressed to the secondary station, the chip enters the receive routine, executes only a few lines of code (address comparison) and returns to perform other tasks. This interrupt will not occupy the CPU for more than two data byte periods (43 microseconds at 375 Kbps). At the end of the frame, the BOV bit is set by the SIU, and the SIU interrupt occurs. In the SIU interrupt service routine,
In the receive interrupt service routine (see Figure 23), counter 0 is turned off, important registers are saved, receive buffer starting address and receive buffer length of the external RAM are set (do not confuse the external RAM settings with that of the internal RAM buffer.)

After reception of an opening flag, the byte processor jumps to the ADDRESS state and waits until the bit processor processes and moves the receiving address byte to SR. Then, the byte processor is triggered to execute the state. In the secondary stations, the CPU monitors the SIUST register for the ADDRESS state (SIUST = 08H). When the ADDRESS state is reached, the byte processor is moved to the next state (CONTROL state), and the ADDRESS state is skipped. Therefore, when the address byte is moved to SR, the byte processor executes the CONTROL state rather than the ADDRESS state and then jumps to the PUSH-1 state. The execution of the CONTROL state causes the contents of SR (the received address byte) to be loaded into the RCB register.

The CPU checks the contents of RCB with the contents of the STAD (Station Address) register. If they match, the receive routine continues to store the received information bytes in the external RAM buffer; otherwise, the byte processor is moved to the very last state (BOV-LOOP), and the program returns from the routine to perform other tasks. The byte processor executes the BOV-LOOP state in each byte boundary until the closing flag of the frame is reached. It then sets the BOV bit and interrupts the CPU (serial interrupt SI set). In the serial interrupt routine the counter 0 is turned back on, and the station is reset back to the receive mode (RBE set).

In Normal operation, in the ADDRESS state, the received address byte is automatically compared with the station address. If they match, the byte processor executes the remaining states; otherwise, the byte processor goes into the idle mode (SIUST = 01H) and waits for the opening flag of the next frame. In the expanded operation, this state is skipped to avoid idle mode. If the byte processor went into the idle mode, clocks which run the byte processor would be turned off, and the byte processor can not be moved to any other states by the CPU. When the byte processor is in idle mode, counter 0 can not be turned on immediately because counter interrupt occurs on the same frame, and program returns to the receive routine and stays there.

If the address byte matches the station address, the byte processor is moved to the CONTROL state again. This time, after execution of the CONTROL state the contents of RCB are the received control byte.

CPU investigates the type of received frame by checking the received control byte. If the receiving frame is not an information frame (i.e. Supervisory frame), execution of receive routine will be terminated to free the
CPU. In Auto mode, the SIU checks the control byte and responds automatically in response to the supervisory frame.

After the control byte is received, it is saved in the stack. The byte processor is moved to the CONTROL state so that the next incoming byte will also be loaded into the RCB register. The byte processor remains in CONTROL state until a byte is processed by the bit processor and moved to SR. The byte processor is then triggered to move the contents of SR to the RCB register. The CPU monitors SIUST and waits until the first Information byte is loaded into the RCB register.

When byte processor reaches the PUSH-1 state (SIUST = 18H), RCB contains the first Information byte. The byte is moved to external RAM (receive buffer), and the byte processor is moved back to the CONTROL state. The process continues until all of the Information bytes are received. When all the Information bytes are received, the program returns from the routine. The byte processor automatically goes through the remaining states, updates the STS register, and interrupts the CPU as it would in Normal operation.

### 6.2.6 TRANSMIT SUBROUTINE

The transmit subroutine codes can be found in the primary and the secondary software (Appendix B). The transmit subroutines of the Primary and secondary stations are identical. A call to transmit routine is made when the RTS and TBF bits of the STS register are set. In Auto mode, RTS is set automatically upon reception of a poll-frame (poll bit of the control byte is set).

In the transmit routine (see Figure 15), the starting address and the transmit buffer length of the external buffer are set. Then the CPU monitors the SIUST register for CONTROL state (SIUST = A8H). In the CONTROL state the bit processor transmits the control byte, while the byte processor goes into the standby mode after it has moved the contents of a location in the internal RAM addressed by the contents of Transmit Buffer Start (TBS) register to the RB register.

While the control byte is being transmitted and the byte processor is in standby, the CPU moves an Information byte from external RAM to the internal RAM location addressed by TBS. The byte processor is then moved to CONTROL state. This will cause the byte processor, in the next byte boundary, to move the contents of the same location in the internal RAM to the RB register (see transmit state diagram.)

When this byte is being transmitted, the byte processor jumps to the DMA-LOOP state (SIUST = B0H) and waits. When the DMA-LOOP state is reached (CPU monitors SIUST for B0H), the CPU loads the next Information byte into the same location in the internal RAM and moves the byte processor to the CONTROL state before it gets to execute the DMA-LOOP state. Note that the same location in the internal RAM is used to transmit the subsequent Information bytes.

When all the Information bytes from the external RAM are transmitted, the byte processor is free to go through the remaining states so that it will transmit the FCS bytes and the closing flag.

### 7.0 CONCLUSIONS

The RUPI, with addition of only a few bytes of code, can accept and transmit large frames with some compromise in the maximum data rate. It can be used in Auto or Flexible mode, with external or internal clocking, automatic CRC checking, and zero bit insertion/deletion. In addition, almost all of the internal RAM is available to be used as general purpose registers, or in conjunction with the external RAM as transmit and receive buffers.

All in all, this feature opens up new areas of applications for this device. Besides transmitting/receiving long frames, it may now be possible to perform arithmetic operations or bit manipulation (e.g. data scrambling) while transmission or reception is taking place, resulting in high throughput. Transmission of continuous flags and transmission with no zero insertion are also possible.

In addition to unlimited frame size, an on-chip controller, automatic SDLC responses, full support of SDLC protocol, 192 bytes of internal RAM, and the highest data rate in self clocked mode compared to other chips make this product very attractive.
APPENDIX A
LISTING OF SOFTWARE MODULES
FOR APPLICATION EXAMPLE 1

$DEBUG NOMOD51
$INCLUDE (REG44.PDF)

; ASSEMBLY CODE FOR PRIMARY STATION (POINT TO POINT)
; FLEXIBLE MODE; PCS OPTION

ORG 00H
SJMP INIT ; LOCATIONS 00 THRU 26H ARE USED
ORG 0BH ; BY INTERRUPT SERVICE ROUTINES.
JMP REC ; VECTOR ADDRESS FOR TIMERS INT.
ORG 23H ; VECTOR ADDRESS FOR SIU INT.
SJMP SIINT

************ INITIALIZATION ************

ORG 26H
INIT: MOV SMD,#00000110B ; EXT CLOCK; PFS=NB=1
MOV TBL,#10H ; INT TRANSMIT BUFFER LENGTH
MOV 20H,#55H ; STATION ADDRESS
MOV TMOD,#00000111B ; COUNTER FUNCTION: MODE 3
MOV IE,#10010010B ; EA=1; SI=1; ETO=1
MOV STS,#11100000B ; TRANSMIT A FRAME

DOT: SJMP DOT ; WAIT FOR AN INTERRUPT

; SIU TRANSMITS THE PFS BYTES, THE OPENING FLAG, THE CONTENTS
; OF LOCATION 20H, THE CALCULATED FCS-BYTES, AND THE CLOSING
; FLAG. AT THE END OF TRANSMISSION, SIU INTERRUPT OCCURS.

SERIAL CHANNEL INTERRUPT ROUTINE

SIINT: CLR SI
JNB RBE,RECVED ; TRANSMITTED A FRAME?

MOV TLO,#0F8H ; YES, INITIALIZE COUNTER REGISTER
MOV DPTR,#200H ; EXT RAM RECEIVE BUFFER START
SETB TRO ; EXT RAM RECEIVE BUFFER LENGTH

RECVED: MOV STS,#11100000B ; TRANSMIT A FRAME
RETI ; RETURN

RECVED: MOV STS,#11100000B ; TRANSMIT A FRAME
RETI ; RETURN

************ RECEIVE INTERRUPT ROUTINE ************

REC: CLR TRO ; DISABLE THE COUNTER 0 INTERRUPT
MOV A,#18H
WAIT1: CJNE A,SIUST,WAIT1
NEXT1: MOV SIUST,#0E8H ; MOVE BVP TO CONTROL STATE
MOV A,#18H
WAIT2: CJNE A,SIUST,WAIT2
MOV A,RCH ; MOVE RECEIVED BYTE INTO ACC.
MOV DPTR,A ; MOVE DATA TO EXT. RAM
INC DPTR ; INCREMENT POINTER TO EXT RAM
LDNz R5,NEXT1 ; LAST BYTE RECEIVED?
RETI ; YES, RETURN

END

292019-28

292019-29
$DEBUG NOMOD51
$INCLUDE (REG44.PDF)

; ASSEMBLY CODE FOR SECONDARY STATION (POINT TO POINT)
; FLEXIBLE MODE; FCS OPTION

ORG 00H
SJMP INIT
ORG 23H ; VECTOR ADDRESS FOR SIU INT.
SJMP SIINT

;*************** LOAD TRANSMIT BUFFER WITH DATA ***************
ORG 26H
INIT:
MOV DPTR,#200H ; EXT RAM XMIT BUFFER START
MOV R3,#0FFH ; EXT RAM XMIT BUFFER LENGTH
LDRA: MOV A,R3
MOVX @DPTR,A ; LOAD EXT BUFFER WITH FFH,FEH,...
INC DPTR ; INCREMENT POINTER
DJNZ R5,LDRA

;************** INITIALIZATION ************************************

MOV SMOD,#0000010B ; EXT CLOCK; PS-NB=1
MOV R1,#01H
MOV TBL,#01H ; INT RAM XMIT BUFFER LENGTH
MOV RSS,#20H ; INT RAM RECEIVE BUFFER START
MOV RBL,#01H ; INT RAM RECEIVE BUFFER LENGTH
MOV STAD,#55H ; STAD ADDRESS=55H
MOV TCON,#00H ; RESET TCON REGISTER
MOV IE,#10010000B ; ENABLE SI INT.; EA=1
MOV IP,#0FFH ; ALL INTERRUPTS; PRIORITY 1
MOV STS,#01000000B ; RBE=1, RECEIVE A FRAME.

DOT: SJMP DOT ; WAIT FOR AN INTERRUPT

; SIU INTERRUPT OCCURS AT THE END OF A RECEIVED FRAME OR
; A TRANSMITTED FRAME.
;************* SERIAL CHANNEL INTERRUPT ROUTINE *************

SIINT: CLR SI
JB RBE,RETN ; RECEIVED A FRAME?
MOV A,R3
CJNE A,20H,HMACH
ACALL TRAN
RETN: RETI ; RECEIVE A FRAME

; TRANSMIT SUBROUTINE IS CALLED TO TRANSMIT A LONG FRAME.
; AFTER TRANSMISSION, SI IS SET. SIU INTERRUPT IS SERVICED
; AFTER THE CURRENT ROUTINE (SIINT) IS COMPLETED.

HMACH: SETB RBE
RETN: RETI

;*************** TRANSMIT SUBROUTINE **********************

TRAN: MOV DPTR,#200H ; EXT RAM RECEIVE BUFFER START
MOV R5,#0FFH ; EXT RAM RECEIVE BUFFER LENGTH
SETB TBF ; SET TRANSMIT BUFFER FULL
SETB RTS ; ENABLE XMISSION OF AN I-FRAME
LOOP: MOVX A,@DPTR ; MOVE THE 1ST I-BYTE INTO ACC.
MOV A,R1
INC DPTR ; INCREMENT POINTER TO EXT. RAM
DJNZ R5,LOOP
NLX A,5
WAILT: CJNE A,SIUST,WAITL ; WAIT FOR XMISSION OF AN I-FRAME
INC DPTR
DJNZ R5,NEXTI
NLX A,5
MOVX A,@DPTR ; ALL BYTES XMITTED?
MOV A,R1
MOV SIUST,#57H ; MOVE DATA INTO INT. RAM @ (TBS)
MOVX @R1,A
MOV SIUST,#57H ; MOVE BYP TO CONTROL STATE
MOVX @R5,20H ; THE SIU TRANSMITS THE FCS-BYTES
DJNZ R5,LOOP
RET
NEXTI: MOV SIUST,#57H ; MOVE BYP TO CONTROL STATE (ASH).
JMP LOOP ; TRANSMIT THE NEXT BYTE

END

292019-30

4-46
FLEXIBILITY IN FRAME SIZE WITH THE 8044

APPENDIX B
LISTING OF SOFTWARE MODULES
FOR APPLICATION EXAMPLE 2

$DEBUG NOMODS1
$INCLUDE (REG44.PDF)

; ASSEMBLY CODE FOR PRIMARY STATION (MULTIPOINT)
; FLEXIBLE MODE; FCS OPTION

ORG 00H
SJMP INIT
JMP REC
ORG 23H
SJMP 2HINT

;********************** LOAD TRANSMIT BUFFER WITH DATA ***************
ORG 26H

INIT:
MOV DPTR,#200H
MOV R3,#0F0H
INC DPTR
SJMP OOH

;******************** INITIALIZATION **********************

; LOCATIONS 00 THRU 26H ARE USED
; BY INTERRUPT SERVICE ROUTINES.
; VECTOR ADDRESS FOR SIU INT.

; INCREMENT POINTER

; LOAD BUFFER WITH FFH,FEH,...00

; EXT RAM XMIT BUFFER

; EXT RAM XMIT BUFFER LENGTH

; LOOP:

; MOVE 0 INTO RAM ADDRESS BY R0

; PUT ZEROS INTO INT. RAM FROM BFH TO 40H.

; MOVE 0 INTO RAM ADDRESS BY R0

; NS COUNTER FOR STAD=55

; NS COUNTER FOR STAD=55

; NS COUNTER FOR STAD=44

; NS COUNTER FOR STAD=44

; POINTER TO SECONDARY STATIONS

; INT. CLKED @ 375K; NRZI-1; PFS-1

; INT. RAM RECEIVE BUFFER START=10H

; INT. RAM RECEIVE BUFFER LENGTH=0

; INT. RAM XMIT BUFFER START=20H

; EXT. RAM XMIT BUFFER LENGTH=1

; EXT. RAM XMIT BUFFER LENGTH=1

; NS-NS=0

; COUNTER FUNCTION, MODE 3

; ADDRESS BYTE=55H

; ADDRESS BYTE=55H

; RBE=RTS=1

; TRANSMIT A LONG FRAME WITH POLL BIT SET, WAIT FOR A
; RESPONSE.

ACALL TRAN

; CALL TRANSMIT ROUTINE

SJMP DOT

; WAIT FOR AN INTERRUPT
FLEXIBILITY IN FRAME SIZE WITH THE 8044

;*************** SERIAL INTERRUPT ROUTINE ***************
SIINT: CLR SI ; CLEAR SI
JB RBI,RETURN ; RECEIVED A FRAME?
MOV A,RS3 ; YES, LOAD ACC WITH REC CTRL BYTE
JB ACC.O,GETI ; IS IT AN I-FRAME?
MOV A,601H ; YES
CJNE A,3AH,Skip
MOV A,30H ; MOVE NS INTO ACC.
INC A ; INCREMENT NS
ANL A,00000011B ; MASK OUT THE LEAST 3 SIG. BITS
MOV 30H,A ; SAVE NS
MOV A,31H ; MOVE NR INTO ACC.
INC A ; INCREMENT NR
ANL A,00000011B ; MASK OUT THE LEAST 3 SIG. BITS
MOV 31H,A ; SAVE NR
RL A ; SHIFT 4 BITS TO LEFT
RL A
RL A
ORL A,30H ; MOVE NS COUNT TO ACC.
RL A ; SHIFT 1 BIT TO LEFT
ORL A,00010000B ; SET THE POLL BIT
MOV TCB,A ; MOVE CONTROL BYTE INTO TCB REG.
MOV STAD,#55H ; TCB: NS2,NS1,NS0,0
MOV 34H,#00H ; GETI
JMP GETI

;*************** RECEIVE INTERRUPT ROUTINE ***************

Rec: CLR TR0 ; TURN OFF COUNTER 0
MOV DPTR,#400H ; EXT. RAM RECEIVE BUFFER START.
MOV R5,#0FFH ; EXT. RAM RECEIVE BUFFER LENGTH
MOV A,18H ; PUSH-1 STATE
Wait1: CJNE A,SIUST,Wait1 ; WAIT FOR THE CONTROL BYTE
PUSH RCB ; SAVE RECEIVE CONTROL BYTE
Nexti: MOV SIUST,#0FFH ; PUSH "BY" INTO CONTROL STATE(10H).
MOV A,18H ; PUSH-1 STATE
Wait2: CJNE A,SIUST,Wait2 ; WAIT FOR AN I-BYTE
MOV A,RCB ; MOVE RECEIVED I-BYTE INTO ACC.
MO VX @DPTR,A ; MOVE DATA TO EXT. RAM
INC DPTR ; INCREMENT PTR TO EXT. RAM
DJNZ R5,Nexti ; IS IT THE LAST I-BYTE?
POP RCB ; YES, RESTORE THE CONTENTS OF RCB
RETI ; RETURN

;*************** TRANSMIT SUBROUTINE ***************

Trans: MOV DPTR,#200H ; EXT. RAM TRANSMIT BUFFER START
MOV R5,#0FFH ; EXT. RAM TRANSMIT BUFFER LENGTH
MOV A,#0AH ; CONTROL STATE
Wait: CJNE A,SiUST,Wait ; WAIT FOR CTRL BYTE XMISSION
MOVX @DPTR,A ; MOVE DATA FROM EXT. RAM TO ACC.
INC DPTR ; INCREMENT POINTER
Djnz R5,Nexti ; IS IT THE LAST I-BYTE?
MOV SIUST,#57H ; NO, XMIT THE LAST I-BYTE
PUSH RCB ; RET
RETI ; RETURN
Nexti: MOV SIUST,#57H ; KEEP "BY" IN CONTROL STATE(18H).
MOV A,#0BH ; DMA-LOOP STATE
JMP Wait ; XMIT THE NEXT BYTE
End
$DEBUG N0MOD51
$INCLUD3 (REG44.PDF)

; ASSEMBLY CODE FOR SECONDARY STATIONS (MULTIPOINT)
; AUTO MODE; FCS OPTION

ORG 00H
BMIJMP INIT
ORG 08H ; VECTOR ADDRESS FOR TIMERO INT.
JMP REC
ORG 10H ; VECTOR ADDRESS FOR EXT. INT. 1
JMP XINT1
ORG 20H ; VECTOR ADDRESS FOR SIU INTERRUPT
JMP SINT

;***************INITIALIZATION ***************

ORG 26H

INIT: MOV SMD,#11010100B ; INT. CLKED @ 375K; NRZI=1; PFS=1
MOV STAD,#55H ; STATION ADDRESS; STAD=44H FOR THE
OTHER STATION
MOV RBS,#10H ; INT. RAM RECEIVE BUFFER START
MOV RBL,#00H ; INT. RAM RECEIVEBUFFER LENGTH
MOV R1,#20H
MOV TBS,R1 ; INT. RAM XMIT BUFFER START
MOV TBL,#10H ; INT. RAM XMIT BUFFER LENGTH
MOV NSNR,#00H
MOV TCON,#00001010B
MOV IE,#00000110B ; TIMER 0: PRIORITY 1
MOV TMOD,#000011B ; COUNTER FUNCTION: NODE 3
MOV STS,#01000010B ; RECEIVE I-FRAME.
MOV TLO,#00FH ; SET COUNTER TO OVERFLOW
MOV TO,#00000010B ; EXT. INT.: EDGE TRIGGERED
MOV SI,#00001101B ; SI=1; ETO=1; EXO=1
MOV IP,#00000010B ; TIMER 0: PRIORITY 1
MOV TCON,#00000000B

; CPU IS INTERRUPTED AT THE END OF RECEPTION (SI SET), AND AT
; THE END OF LONG-FRAME TRANSMISSION (EXO SET).

DOT: BMIJMP DOT ; WAIT FOR AN INTERRUPT.

; CPU IS INTERRUPTED AT THE END OF RECEPTION (SI SET), AND AT
; THE END OF LONG-FRAME TRANSMISSION (EXO SET).

XINT1: SETB P1.7 ; DISABLE CTS PIN
RETI ; RETURN.

;*************** SERIAL INTERRUPT ROUTINE ***************

SINIT: CLR SI
JB AM,HOP ; ADDRESS MATCHED?
CLR EA ; DISABLE ALL INTERRUPTS
MOV STS,#01000010B ; RBE=1; NS=1
MOV TLO,#00FH
SETB TRO ; TURN ON COUNTER 0
SETB EA ; ENABLE ALL INTERRUPTS
RETI ; RETURN.

; HOP: JB TBF,GETI ; A FRAME TRANSMITTED?
SETB TBF ; ENABLE TRANSMISSION OF I-FRAME
CLR P1.7 ; ENABLE CTS PIN
ACALL TRAN ; CALL TRANSMIT ROUTINE
GETI: JB RBE,RETURN ; A FRAME RECEIVED?
CLR EA ; DISABLE ALL INTERRUPTS
SETB RBE ; PUT RPI IN RECEIVE MODE
MOV TLO,#00FH ; RET.
SETB TRO ; TURN ON COUNTER 0
SETB EA ; ENABLE ALL INTERRUPTS
RETI ; RETURN.

;*************** TRANSMIT SUBROUTINE ***************

TRAN: MOV DPTR,#200H ; EXT. RAM TRANSMIT BUFFER START
MOV R5,#0FFH ; EXT. RAM TRANSMIT BUFFER LENGTH
MOV A,#0AH ; CONTROL STATE
ACALL CINE ; CINE
MOVX A,@DPTR ; MOVE DATA FROM EXT. RAM TO ACC.
MOV @R1,A ; MOVE DATA INTO INT. RAM AT @R1
INC DPTR ; INCREMENT POINTER
DJNZ R5,NXTI ; IS IT THE LAST I-BYTE ?
MOV SIUST,#57H ; XMIT THE LAST I-BYTE
RET ; RETURN.

NXTI: MOV SIUST,#57H ; KEEP "BYP" IN CONTROL STATE
MOV A,#0AH ; DMA-LOOP STATE
JMP WAIT ; TRANSMIT THE NEXT BYTE

292019-37

292019-38

292019-39

4-49
FLEXIBILITY IN FRAME SIZE WITH THE 8044

;**********RECEIVE INTERRUPT ROUTINE**************

REC:  CLR TR0  ; TURN OFF COUNTER 0
      MOV DPTR,#200H  ; EXT. RAM RECEIVE BUFFER START
      MOV RS,#0FFH  ; EXT. RAM RECEIVE BUFFER LENGTH
      MOV A,#08H  ; ADDRESS STATE

HOLD: CJNE A,SIUST,HOLD  ; WAIT FOR ADDRESS BYTE
      MOV SIUST,#0EFH  ; MOVE "BYP" INTO CONTROL STATE
      MOV A,#18H  ; SKIP THE ADDRESS STATE

      MOV A,#18H  ; PUSH-1 STATE
       WAIT1: CJNE A,SIUST,WAIT1  ; WAIT FOR THE ADDRESS BYTE
      MOV A,RCB  ; MOVE THE RECEIVED ADDRESS BYTE TO ACC.
      CJNE A,STAD,WAIT2  ; ADDRESS MATCHED?
      SJMP WAIT3  ; YES.

WAIT2: MOV RCB,#00010000B  ; MOVE INFO. CONTROL BYTE TO RCB
      MOV SIUST,#0CFH  ; MOVE "BYP" INTO BOV-LOOP STATE
      RETI  ; RETURN

      ; WAIT3: MOV SIUST,#0EFH  ; MOVE "BYP" INTO CONTROL STATE
      MOV A,#18H  ; PUSH-1 STATE

WAIT4: CJNE A,SIUST,WAIT4  ; WAIT FOR THE CONTROL BYTE
      MOV A,RCB  ; MOVE RECEIVE CONTROL BYTE INTO ACC.
      JB ACC.0,RTRN  ; IF NOT AN I-BYTE RETURN
      PUSH RCB  ; SAVE RECEIVE CONTROL BYTE

      PUSH SIUST  ; PUSH "BYP" INTO CONTROL STATE(10H).

NEXTI: MOV SIUST,#0EFH  ; MOVE "BYP" INTO CONTROL STATE(10H).
      MOV A,#18H  ; PUSH-1 STATE

WAIT5: CJNE A,SIUST,WAIT5  ; WAIT FOR AN I-BYTE
      MOV A,RCB  ; MOVE RECEIVED I-BYTE INTO ACC.
      MOVX @DPTR,A  ; MOVE DATA TO EXT. RAM
      INC DPTR  ; INCREMENT PTR TO EXTERNAL RAM
      DJNE R5,NEXTI  ; IS IT THE LAST I-BYTE?
      POP RCB  ; YES. RESTORE THE CONTENTS OF RCB

      RTRN: RETI  ; RETURN

END

292019-40
Using the 80186/188/C186/C188
1.0 INTRODUCTION

The 80186 microprocessor family holds the position of industry standard among high integration microprocessors. VLSI technology incorporates the most commonly used peripheral functions with a 16-bit CPU on the same silicon die to assure compatibility and high reliability (see Figure 1). The 80186 reputation for flexibility and uncomplicated programming make it the first choice microprocessor for such data control applications as local area network equipment, PC add-on cards, terminals, disk storage subsystems, avionics, and medical instrumentation.

There are two purposes to this Application Note. The first is to explain the operation of the integrated 80186 peripheral set with a degree of detail not possible in the data sheet. The second is to describe, through examples, the use of the 80186 with other digital logic such as memory.

The 80186 family actually consists of 4 devices: the original 80186 and 80188, and the new 80C186 and 80C188 microprocessors manufactured on Intel's CHMOS III process. The 80188 and 80C188 are 16-bit microprocessors but have 8-bit external data buses. The 80C186 and 80C188 offer the advantage of increased speed (up to 16 MHz) and important new features including a Refresh Control Unit, Power-Save Logic, and ONCETM Mode (see Figure 2). For simplicity, this Ap Note uses the name 80186 to refer collectively to all the members of the 80186 family. Differences between individual processors are pointed out as necessary.
2.0 OVERVIEW OF THE 80186 FAMILY

2.1 The CPU

The 80186 CPU shares a common base architecture with the 8086, 8088, 80286, and 80386 processors. It is completely object code compatible with the 8086/88. This architecture features four 16-bit general purpose registers (AX, BX, CX, DX) which may be used as operands in most arithmetic operations in either 8- or 16-bit units. It also features four 16-bit pointer registers (SI, DI, BP, SP) which may be used both in arithmetic operations and in accessing memory based variables. Four 16-bit segment registers (CS, DS, SS, ES) allow simple memory partitioning to aid construction of modular programs. Finally, it has a 16-bit instruction pointer and a 16-bit status register.

Physical memory addresses are generated by the 80186 identically to the 8086. The 16-bit segment value is shifted left 4 bits and then added to an offset value which is derived from combinations of the pointer registers, the instruction pointer, and immediate values (see Figure 3). Any carry of this addition is ignored. The result is a 20-bit physical address.

The 80186 has a 16-bit ALU which performs 8 or 16-bit arithmetic and logical operations. It provides for data movement among registers, memory and I/O space. In addition, the CPU allows for high speed data transfer from one area of memory to another using string move instructions, and to or from an I/O port and memory using block I/O instructions. Finally, the CPU provides many conditional branch and control instructions.

In the 80186, as in the 8086, instruction fetching and instruction execution are performed by separate units: the bus interface unit and the execution unit, respectively. The 80186 also has a 6-byte prefetch queue as does the 8086. The 80188 has a 4-byte prefetch queue as does the 8088. As a program is executing, opcodes are fetched from memory by the bus interface unit and placed in this queue. Whenever the execution unit requires another opcode byte, it takes the byte out of the queue. Effective processor throughput is increased by...
adding this queue, since the bus interface unit may continue to fetch instructions while the execution unit executes a long instruction. Then, when the CPU completes this instruction, it does not have to wait for another instruction to be fetched from memory.

2.2 80186 CPU Enhancements

Although the 80186 is completely object code compatible with the 8086, most of the 8086 instructions require fewer clock cycles to execute on the 80186 than on the 8086 because of hardware enhancements in the bus interface unit and the execution unit. In addition, the 80186 has many new instructions which simplify assembly language programming, enhance the performance of high level language implementations, and reduce code size. The added instructions are described in Appendix H of this Ap Note.

2.3 DMA Unit

The 80186 includes a DMA unit which provides two flexible DMA channels. This DMA unit will perform transfers to or from any combination of I/O space and memory space in either byte or word units. Every DMA cycle requires two to four bus cycles, one or two to fetch the data and one or two to deposit the data. This allows word data to be located on odd boundaries, or byte data to be moved from odd locations to even locations.

Each DMA channel maintains independent 20-bit source and destination pointers. Each of these pointers may independently address either I/O or memory space. After each DMA cycle, the pointers may be optionally incremented, decremented, or maintained constant. Each DMA channel also maintains a transfer count which can terminate a series of DMA transfers after a pre-programmed number of transfers.

2.4 Timers

The timer unit contains 3 independent 16-bit timer/counters. Two of them can count external events, provide waveforms based on either the CPU clock or an external clock, or interrupt the CPU after a specified count. The third timer/counter counts only CPU clocks. After a programmable interval, it can interrupt the CPU, provide a clock pulse to either or both of the other timer/counters, or send a DMA request pulse to the integrated DMA controller.

2.5 Interrupt Controller

The integrated interrupt controller arbitrates interrupt requests between all internal and external sources. It can be directly cascaded as the master to an external 8259A or 82C59A interrupt controller. In addition, it can be configured as a slave controller.

2.6 Clock Generator

The on-board crystal oscillator can be used with a parallel resonant, fundamental mode crystal at 2X the desired CPU clock speed (i.e., 16 MHz for an 8 MHz 80186), or with an external oscillator also at 2X the CPU clock. The output of the oscillator is internally divided by two to provide the 50% duty cycle CPU clock from which all 80186 system timing is derived. The CPU clock is externally available, and all timing parameters are referenced to it.

2.7 Chip Select and Ready Generation Unit

The 80186 includes integrated chip select logic which can be used to enable memory or peripheral devices. Six output lines are used for memory addressing and seven output lines are used for peripheral addressing.

The memory chip select lines are split into 3 groups for separately addressing the major memory areas in a typical 80186 system: upper memory for reset ROM, lower memory for interrupt vectors, and mid-range memory for program memory. The size of each of these regions is user programmable. The starting location and ending location of lower memory and upper memory are fixed at 00000H and FFFFFFFH respectively; the starting location of the mid-range memory is user programmable.

Each of the seven peripheral select lines address one of seven contiguous 128 byte blocks above a programmable base address. This base address can be located in
either memory or I/O space so that peripheral devices may be I/O or memory mapped.

Each of the programmed chip select areas has associated with it a set of programmable ready bits. These bits allow a programmable number of wait states (0 to 3) to be automatically inserted whenever an access is made to the area of memory associated with the chip select area. In addition, a bit determines whether the external ready signals (ARDY and SRDY) will be used, or whether they will be ignored (i.e., the bus cycle will terminate even though a ready has not been returned on the external pins). There are 5 total sets of ready bits which allow independent ready generation for each of upper memory, lower memory, mid-range memory, peripheral devices 0–3 and peripheral devices 4–6.

2.8 Integrated Peripheral Accessing

The integrated peripheral and chip select circuitry is controlled by sets of 16-bit registers accessed using standard input, output, or memory access instructions. These peripheral control registers are all located within a 256 byte block which can be placed in either memory or I/O space. Because they are accessed exactly as if they were external devices, no new instruction types are required to access and control the integrated peripherals.

3.0 USING THE 80186 FAMILY

3.1 Bus Interfacing to the 80186

3.1.1 OVERVIEW

The 80186 bus structure is very similar to that of the 8086. It includes a multiplexed address/data bus, along with various control and status lines (see Table 1). Each bus cycle requires a minimum of 4 CPU clock cycles along with any number of wait states required to accommodate access limitations of external memory or peripheral devices. The bus cycles initiated by the 80186 CPU are identical to the bus cycles initiated by the 80186 integrated DMA unit.

Each clock cycle of the 80186 bus cycle is called a “T” state, and are numbered sequentially T₁, T₂, T₃, T₄ and T₅. Additional idle T states (T₆) can occur between T₄ and T₁ when the processor requires no bus activity (instruction fetches, memory writes, I/O reads, etc.). The ready signals control the number of wait states (TW) inserted in each bus cycle. The maximum number of wait states is unbounded.

The beginning of a T state is signaled by a high to low transition of the CPU clock. Each T state is divided into two phases, phase 1 (or the low phase) and phase 2 (or the high phase) (see Figure 4).

Different types of bus activity occur for all of the T-states (see Figure 5). Address generation information occurs during T₁, data generation during T₂, T₃, T₄ and T₅. The beginning of a bus cycle is signaled by the status lines of the processor going from a passive state (all high) to an active state in the middle of the T-state immediately before T₁ (either a T₄ or a T₅). Information concerning an impending bus cycle appears during the T-state immediately before the first T-state of the cycle itself. Two different types of T₄ and T₅ can be generated: one where the T state is immediately followed by a bus cycle, and one where the T state is immediately followed by an idle T state.

During the first type of T₄ or T₅, status information concerning the impending bus cycle is generated for the bus cycle immediately to follow. This information will be available no later than tCHSV after the low-to-high transition of the 80186 clock in the middle of the T state. During the second type of T₄ or T₅, the status outputs remain inactive because no bus cycle will follow. The decision on which type T₄ or T₅ state to present is made at the beginning of the T-state preceding the T₄ or T₅ state (see Figure 6). This determination has an effect on bus latency (see Section 3.3.2).
Figure 5. Example Bus Cycle of the 80186

Table 1. 80186 Bus Signals

<table>
<thead>
<tr>
<th>Function</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>address/data</td>
<td>AD0–AD15</td>
</tr>
<tr>
<td>address/status</td>
<td>A16/S3-A19-S6, BHE/S7</td>
</tr>
<tr>
<td>co-processor control</td>
<td>TEST</td>
</tr>
<tr>
<td>local bus arbitration</td>
<td>HOLD, HLDA</td>
</tr>
<tr>
<td>local bus control</td>
<td>ALE, RD, WR, DT/R, DEN</td>
</tr>
<tr>
<td>multi-master bus</td>
<td>LOCK</td>
</tr>
<tr>
<td>ready (wait) interface</td>
<td>SRDY, ARDY</td>
</tr>
<tr>
<td>status information</td>
<td>S0–S2</td>
</tr>
</tbody>
</table>

Figure 6. Active-Inactive Status Transitions in the 80186
3.1.2 PHYSICAL ADDRESS GENERATION

Physical addresses are generated by the 80186 during T1 of a bus cycle. Since the address and data lines are multiplexed on the same set of pins, addresses must be latched during T1 if they are required to remain stable for the duration of the bus cycle. To facilitate latching of the physical address, the 80186 generates an active high ALE (Address Latch Enable) signal which can be directly connected to a transparent latch's strobe input.

Figure 7 illustrates the physical address generation parameters of the 80186. Addresses are guaranteed valid no greater than tCLAV after the beginning of T1, and remain valid at least tCLAX after the end of T1. The ALE signal is driven high in the middle of the T state (either T4 or T5) immediately preceding T1 and is driven low in the middle of T1, no sooner than tAVLL after addresses become valid. This parameter (tAVLL) is required to satisfy the address latch set-up times of address valid until strobe inactive. Addresses remain stable on the address/data bus at least tLLAX after ALE goes inactive to satisfy address latch hold times.

| 1. tCHLH: Clock high to ALE high |
| 2. tCLAV: Clock low to address valid |
| 3. tCHLL: Clock high to ALE low |
| 4. tCLAX: Clock low to address invalid (address hold from clock low) |
| 5. tAVLL: ALE low to address invalid (address hold from ALE) |
| 6. tLAX: Address valid to ALE low (address setup to ALE) |

Figure 7. Address Generation Timing of the 80186

Because ALE goes high before addresses become valid, the delay through the address latches will be the propagation delay through the latch rather than the delay from the latch strobe, which is typically longer than the propagation delay. Note that the 80186 drives ALE high one full clock phase earlier than the 8086 or the 82C88 bus controller, and keeps it high throughout the 8086 or 82C88 ALE high time (i.e., the 80186 ALE pulse is wider).

A typical circuit for latching physical addresses is shown in Figure 8. This circuit uses 3 transparent octal non-inverting latches to demultiplex all 20 address bits provided by the 80186/80188. Typically, the upper 4 address bits only select among various memory components or subsystems, so when the integrated chip selects (see Section 8) are used, these upper bits need not be latched. The worst case address generation time from the beginning of T1 (including address latch propagation time for the circuit is:

\[ t_{CLAV} + t_{PD} \]

Many memory or peripheral devices may not require addresses to remain stable throughout a data transfer. If a system is constructed wholly with these types of devices, addresses need not be latched. In addition, two of the peripheral chip select outputs of the 80186 may be configured to provide latched A1 and A2 outputs for peripheral register selects in a system which does not demultiplex the address/data bus.

Figure 8. Demultiplexing the Address Bus of the 80186 Using Transparent Latches
One more signal is generated by the 80186 to address memory: BHE (Bus High Enable). This signal, along with A0, is used to enable byte devices connected to either or both halves (bytes) of the 16-bit data bus. Because A0 is used only to enable devices onto the lower half of the data bus, memory chip address inputs are usually driven by address bits A1–A19, not A0–A19. This provides 512K unique word addresses, or 1M unique byte addresses. BHE is not present on the 8-bit 80188. All data transfers occur on the 8-bits of the data bus.

**3.1.3 80186/80C186 DATA BUS OPERATION**

Throughout T₂, T₃, T₄ and T₅ of a bus cycle the multiplexed address/data bus becomes a 16-bit data bus. Data transfers on this bus may be either bytes or words. All memory is byte addressable (see Figure 9).

All bytes with even addresses (A₀ = 0) reside on the lower 8 bits of the data bus, while all bytes with odd addresses (A₀ = 1) reside on the upper 8 bits of the data bus. Whenever an access is made to only the even byte, A₀ is driven low, BHE is driven high, and the data transfer occurs on D₀–D₇ of the data bus. Whenever an access is made to only the odd byte, BHE is driven low, A₀ is driven high, and the data transfer occurs on D₈–D₁₅ of the data bus. Finally, if a word access is performed to an even address, both A₀ and BHE are driven low and the data transfer occurs on D₀–D₁₅ of the data bus.

Word accesses are made to the addressed byte and to the next higher numbered byte. If a word access is performed to an odd address, two byte accesses must be performed, the first to access the odd byte at the first word address on D₈–D₁₅, the second to access the even byte at the next sequential word address on D₀–D₇. For example, in Figure 9, byte 0 and byte 1 can be individually accessed in two separate bus cycles to byte addresses 0 and 1 at word address 0. They may also be accessed together in a single bus cycle to word address 0. However, if a word access is made to address 1, two bus cycles will be required, the first to access byte 1 at word address 0 (note byte 0 will not be accessed), and the second to access byte 2 at word address 2 (note byte 3 will not be accessed). This is why all word data should be located at even addresses to maximize processor performance.

When byte reads are made, the data returned on the unused half of the data bus is ignored. When byte writes are made, the data driven on the unused half of the data bus is indeterminate.

**3.1.4 80188/80C188 DATA BUS OPERATION**

Because the 80188 and 80C188 externally have only 8-bit data buses, the above discussion about upper and lower bytes of the data bus does not apply. No performance improvement will occur if word data is placed on even boundaries in memory space. All word accesses require two bus cycles, the first to access to lower byte of the word; the second to access the upper byte of the word.

Any 80188/80C188 access to the integrated peripherals is performed 16 bits at a time, whether byte or word addressing is used. If a byte operation is used, the external bus only indicates a single byte transfer even though the word access takes place.

**3.1.5 GENERAL DATA BUS OPERATION**

Because of the bus drive capabilities of the 80186, additional buffering may not be required in many small systems. If data buffers are not used in the system, care should be taken not to allow bus contention between the 80186 and the devices directly connected to the 80186 data bus. Since the 80186 floats the address/data bus before activating any command lines, the only requirement on a directly connected device is that it float its output drivers after a read before the 80186 begins to drive address information for the next bus cycle. The parameter of interest here is the minimum time from RD inactive until addresses active for the next bus cycle (tRHAV). If the memory or peripheral device cannot disable its output drivers in this time, data buffers will be required to prevent both the 80186 and the device from driving these lines concurrently. This parameter is unaffected by the addition of wait states. Data buffers solve this problem because their output float times are typically much faster than the 80186 required minimum.
If data buffers are required, the 80186 provides DEN (Data ENable) and DT/R (Data Transmit/Receive) signals to simplify buffer interfacing. The DEN and DT/R signals are activated during all bus cycles. The DEN signal is driven low whenever the processor is either ready to receive data (during a read) or when the processor is ready to send data (during a write). In other words, DEN is low during any active bus cycle when address information is not being generated on the address/data pins. In most systems, the DEN signal should not be directly connected to the OE input of buffers, since unbuffered devices (or other buffers) may be directly connected to the processor's address/data pins. If DEN were directly connected to several buffers, contention would occur during read cycles, as many devices attempt to drive the processor bus. Rather, it should be a factor (along with the chip selects for buffered devices) in generating the output enable.

The DT/R signal determines the direction of data through the bi-directional buffers. It is high whenever

**3.1.6 CONTROL SIGNALS**

The 80186 directly provides the control signals RD, WR, LOCK and TEST. In addition, the 80186 provides the status signals S0–S2 and S6 from which all other required bus control signals can be generated.

### 3.1.6.1 RD and WR

The RD and WR signals strobe data to or from memory or I/O space. The RD signal is driven low at the beginning of T2, and is driven high at the beginning of T4 during all memory and I/O reads (see Figure 11). RD will not become active until the 80186 has ceased driving address information on the address/data bus. Data is sampled into the processor at the beginning of T4. RD will not go inactive until the processor’s data hold time has been satisfied.
Note that the 80186 does not provide separate I/O and memory RD signals. If separate I/O read and memory read signals are required, they can be synthesized using the S2 signal (which is low for all I/O operations and high for all memory operations) and the RD signal (see Figure 12). It should be noted that if this approach is used, the S2 signal will require latching, since the S2 signal (like S0 and S1) goes to an inactive state well before the beginning of T4 (where RD goes inactive). If S2 was directly used for this purpose, the type of read command (I/O or memory) could change just before T4 as S2 goes to the inactive state (high). The status signals may be latched using ALE the same as the address signals (often using the spare bits in the address latches).

Often the lack of a separate I/O and memory RD signal is not important in an 80186 system. Each 80186 chip select signal will respond to accesses exclusively in memory or I/O space. Thus, when a chip select is used, the external device is enabled only during accesses to the proper address in the proper space.

The WR signal is also driven low at the beginning of T2 and driven high at the beginning of T4 (see Figure 13). In similar fashion to the RD signal, the WR signal is active for all memory and I/O writes. Again, separate memory and I/O control lines may be generated using the latched S2 signal along with WR. More important, however, is the role of the active-going edge of WR. At the time WR makes its high-to-low transition, valid write data is not present on the data bus. This has consequences when using WR to enable such devices as DRAMs since those devices require the data to be stable on the falling edge. In DRAM applications, the problem is solved by the DRAM controller (an Intel 8207, for example). For other applications which require valid data before the WR transition, place cross-coupled NAND gates between the CPU and the device on the WR line (see Figure 14). The added gates delay the active-going edge of WR to the device by one clock phase, at which time valid data is driven on the bus by the 80186.
If the RD line is externally grounded during reset and remains grounded during processor operation, the 80186 will enter Queue Status Mode. When in this mode, the WR and ALE signals become queue status outputs, reflecting the status of the internal prefetch queue during each clock cycle. These signals are provided to allow a processor extension (such as the Intel 8087 floating point processor) to track execution of instructions within the 80186. The interpretation of QS0 (ALE) and QS1 (WR) is given in Table 2. These signals change on the high-to-low clock transition, one clock phase earlier than on the 8086. Note that since execution unit operation is independent of bus interface unit operation, queue status lines may change in any T state.

Since the ALE, RD, and WR signals are not directly available from the 80186 when it is configured in queue status mode, these signals must be derived from the status lines S0–S2 using an external 82C88 bus controller (see Figure 15). To prevent the 80186 from accidentally entering queue status mode during reset, the RD line is internally provided with a weak pullup device.

Table 2. 80186 Queue Status

<table>
<thead>
<tr>
<th>QS1</th>
<th>QS0</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>no operation</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>first byte of instruction taken from queue</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>queue was reinitialized</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>subsequent byte of instruction taken from queue</td>
</tr>
</tbody>
</table>

NOTES:
1. tCLOV: Clock low until data valid
2. tCVCTV: Clock low until WR active
3. tCVCTX: Clock low until WR inactive
4. tCLDOX: Clock high until data invalid
5. tWHDX: WR inactive until data invalid
3.1.6.3 Status Lines

The 80186 provides 3 status outputs which indicate the type of bus cycle currently being executed. These signals go from an inactive state (all high) to one of seven possible active states during the T state immediately preceding \( T_1 \) of a bus cycle (see Figure 6). The possible status line encodings are given in Table 3. The status lines are driven to their inactive state in the \( T_3 \) or \( T_W \) state immediately preceding \( T_4 \) of the current bus cycle.

Table 3. 80186 Status Line Interpretation

<table>
<thead>
<tr>
<th>( S_2 )</th>
<th>( S_1 )</th>
<th>( S_0 )</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>interrupt acknowledge</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>read I/O</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>write I/O</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>halt</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>instruction fetch</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>read memory</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>write memory</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>passive</td>
</tr>
</tbody>
</table>

The status lines may be directly connected to an 82C88 bus controller, which provides local bus control signals or multi-bus control signals (see Figure 15). Use of the 82C88 bus controller does not preclude the use of the 80186 generated RD, WR and ALE signals, however. The 80186 directly generated signals can provide local bus control signals, while an 82C88 can provide multi-bus control signals.

The \( S_7 \) and \( BHE \) are logically equivalent signals provided by the 80186 and the 80C186 (see Section 3.1.2). \( S_7 \) is always high on the 80188 and 80C188 (except during 80C188 DRAM refresh cycles) which signifies the presence of an 8-bit data bus.

Three other status signals are available on the 8086 but not on the 80186. They are \( S_3 \), \( S_4 \), and \( S_5 \). Taken together, \( S_3 \) and \( S_4 \) indicate the segment register from which the current physical address has been derived. \( S_5 \) indicates the state of the interrupt flip-flop. On the 80186, these signals will always be low.

3.1.6.4 TEST and LOCK

Finally, the 80186 provides a TEST input and a LOCK output. The TEST input is used in conjunction with the processor WAIT instruction. It is typically driven by a coprocessor to indicate whether it is busy.

The LOCK output is driven low whenever the data cycles of a LOCKED instruction are executed. A LOCKED instruction is generated whenever the LOCK prefix occurs immediately before an instruction. The LOCK prefix is active for the single instruction immediately following the LOCK prefix. The LOCK signal indicates to a bus arbiter (e.g., the 8289) that a series of locked data transfers is occurring. The bus arbiter should under no circumstances release the bus while locked transfers are occurring. The 80186 will not recognize a bus HOLD, nor will it allow DMA cycles to be run by the integrated DMA controller during locked data transfers. LOCKED transfers are typically used in multiprocessor systems to access memory based semaphore variables which control access to shared system resources.

On the 80186, the LOCK signal will go active during \( T_1 \) of the first DATA cycle of the locked transfer. It is driven inactive during \( T_4 \) of the last DATA cycle of the locked transfers (assuming no wait states). On the 8086, the LOCK signal is activated immediately after the LOCK prefix is executed. The LOCK prefix may be executed well before the processor is prepared to perform the locked data transfer. This has the unfortunate consequence of activating the LOCK signal before the first LOCKED data cycle is performed. Since LOCK is active before the 8086 requires the bus for the data transfer, opcode pre-fetching can be LOCKED. LOCKED prefetching will not occur with the 80186.

The LOCK output is also driven low during interrupt acknowledge cycles when the integrated interrupt controller operates in Cascade or Slave Modes (see Sections 6.5.2 and 6.5.3). In these modes, the operation of the LOCK pin may be altered when an interrupt occurs.

Two additional status signals are provided by 80186 family members. \( S_6 \) provides information concerning the unit generating the bus cycle. It is time multiplexed with A19, and is available during \( T_2 \), \( T_3 \), \( T_4 \) and \( T_W \). In the 8086 family, all central processors (e.g., the 8086 and 8087) drive this line low, while all I/O processors (e.g., 8089) drive this line high during their respective bus cycles. Following this scheme, the 80186 drives this line low whenever the bus cycle is generated by the 80186 CPU, but drives it high when the bus cycle is generated by the integrated 80186 DMA unit. This allows external devices to distinguish between bus cycles fetching data for the CPU from those transferring data for the DMA unit.
during execution of a software-LOCKED instruction. See Section 6.5.4 for a description of additional hardware necessary to block DMA and HOLD requests under such circumstances.

### 3.1.7 HALT TIMING

A HALT bus cycle signifies that the 80186 CPU has executed a HLT instruction. It differs from a normal bus cycle in two ways.

The first way a HALT bus cycle differs from a normal bus cycle is that neither RD nor WR will be driven active. Address and data information will not be driven by the processor, and no data will be returned. The second way a HALT bus cycle differs from a normal bus cycle is that the S0–S2 status lines go to their inactive state (all high) during T2 of the bus cycle, well before they go to their inactive state during a normal bus cycle.

Like a normal bus cycle, however, ALE is driven active. Since no valid address information is present, the information strobed into the address latches should be ignored. This ALE pulse can be used, however, to latch the HALT status from the S0–S2 status lines.

The processor being halted does not interfere with the operation of any of the 80186 integrated peripheral units. This means that if a DMA transfer is pending while the processor is halted, the bus cycles associated with the transfer will run. In fact, DMA latency time will improve while the processor is halted because the DMA unit will not be contending with the processor for access to the 80186 (see section 4.4.1).

### 3.1.8 82C88 AND 8289 INTERFACING

The 82C88 and 8289 are the bus controller and multimaster bus arbitration devices used with the 8086. Because the 80186 bus is similar to the 8086 bus, they can be used with the 80186. Figure 16 shows an 80186 interconnection to these two devices.

The 82C88 bus controller generates control signals (RD, WR, ALE, DT/R, DEN, etc.) for an 8086 maximum mode system. It derives its information by decoding status lines S0–S2 of the processor. Because the 80186 and the 8086 drive the same status information on these lines, the 80186 can be directly connected to the 82C88 just as in an 8086 system. Using the 82C88 with the 80186 does not prevent using the 80186 control signals. Many systems require both local bus control signals and system bus control signals. In this type of system, the 80186 lines could be used as the local signals, with the 82C88 lines used as the system signals.

Note that in an 80186 system, the 82C88 generated ALE pulse occurs later than that of the 80186 itself. In many multimaster bus systems, the 82C88 ALE pulse should be used to strobe the addresses into the system bus address latches to insure that the address hold times are met.

The 8289 bus arbiter arbitrates the use of a multi-master system bus among various devices, each of which can become the bus master. This component also decodes status lines S0–S2 directly to determine when the system bus is required. When the system bus is required, the 8289 forces the processor to wait until it has acquired control of the bus, then it allows the processor to drive address, data and control information onto the system bus. The system determines when it requires system bus resources by an address decode. Whenever the address being driven coincides with the address of an on-board resource, the system bus is not required and thus will not be requested. The circuit shown in Figure 17 factors the 80186 chip select lines to determine when the system bus should be requested, or when the 80186 request can be satisfied using a local resource.

### 3.1.9 READY INTERFACING

The 80186 provides two ready lines, a synchronous ready (SRDY) line and an asynchronous ready (ARDY) line. These lines signal the bus controller to insert wait states (Tw) into a CPU bus cycle, allowing slower devices to respond to bus activity. Wait states will only be inserted when both ARDY and SRDY are low, i.e., only one of the lines need be active to terminate a bus cycle. Figure 17 depicts the logical ORing of the ARDY and SRDY functions. Any number of wait states may be inserted into a bus cycle. The 80186 will ignore the RDY inputs during any accesses to the integrated peripheral registers and to any area where the chip select ready bits indicate that the external ready should be ignored.
The timing required by the two RDY lines is different. Inputs to the ARDY pin will be internally synchronized to the CPU clock before being presented to the rest of the bus control logic as shown in Figure 17. The first flip-flop is used to “resolve” the asynchronous transition of the ARDY line. It will achieve a definite high or low level before its output is latched into the second flip-flop. When latched high, it passes along the level present on the ARDY line; when latched low, it forces not ready to be passed along to the rest of the circuit. (See Appendix B for synchronizer information.)

In a Normally-Not-Ready implementation the setup and hold times of both the resolution flip-flop and the ready latch must be satisfied. The ARDY pin must go active at least $T_{ARYHCH}$ (also denoted $T_{ARYCH}$) before the rising edge of $T_2$, $T_3$ or $T_W$, and stay active until $T_{CLRAX}$ after the falling edge of $T_3$ or $T_W$ to stop generation of wait states and terminate the bus cycle. If ARDY goes active before the rising edge of $T_2$ and stays active after the falling edge of $T_3$ there will be no wait state inserted.

In a Normally-Ready implementation the setup and hold times of either the resolution flip-flop or the ready latch must be met. Wait states will be generated if ARDY goes inactive $T_{ARYHCH}$ (also denoted $T_{ARYCH}$) before the rising edge of $T_2$, $T_3$ or $T_W$, and stay active until $T_{CLRAX}$ after the falling edge of $T_3$ or $T_W$ to stop generation of wait states and terminate the bus cycle. If ARDY goes inactive before the rising edge of $T_2$ and stays active after the falling edge of $T_3$ there will be no wait state inserted.

Figure 18 depicts activity for Normally-Ready and Normally-Not-Ready configurations of external logic. Remember that for ARDY to force wait states, SRDY must be low as well.
In a Normally-Not-Ready system, wait states will be inserted unless:
1. \( t_{ARYCH} \) (also denoted \( t_{ARYCH} \)): ARDY active to clock high (ARDY resolution setup time)
2. \( t_{CLRX} \): Clock low to ARDY inactive (ARDY active hold time)

In a Normally-Ready system, wait states will be inserted if:
1. \( t_{ARYCH} \) (also denoted \( t_{ARYCH} \)): ARDY low to clock high (ARDY resolution setup time)
2. \( t_{ARYCHL} \): Clock high to ARDY high (ARDY inactive hold time)

Alternatively, in a Normally-Ready system, wait states will be inserted if:
1. \( t_{ARYCL} \): ARDY low to clock low (ARDY setup time)
2. \( t_{CLRX} \): Clock low to ARDY high (ARDY active hold time)
ARDY must meet \( t_{ARYCL} \) and \( t_{CLRX} \) or undesired CPU operation will result.

Figure 18. ARDY Transitions
before the rising edge of \( T_2 \) and stays inactive a minimum of \( T_{ARYCHL} \) after the edge, or if \( ARDY \) goes inactive at least \( T_{ARYLCL} \) before the falling edge of \( T_3 \) and stays inactive a minimum of \( TCLARX \) after the edge. The 80186 ready circuitry performs this way to allow a slow device the maximum amount of time to respond with a not ready after it has been selected.

The synchronous ready (SRDY) line requires that all transitions on this line during \( T_2 \), \( T_3 \), or \( T_W \) satisfy setup and hold times (\( t_{SRYCL} \) and \( t_{CLSRY} \) respectively). If these requirements are not met, the CPU will not function properly. Valid transitions on this line and subsequent wait state insertion is shown in Figure 19. The bus controller looks at SRDY at the beginning of each \( T_3 \) and \( T_W \). If the line is sampled active at the beginning of either of these two cycles, that cycle will be immediately followed by \( T_4 \). If the line is sampled inactive at the beginning of either \( T \) state, that cycle will be followed by a \( T_W \). Any asynchronous transition on the SRDY line not occurring at the beginning of \( T_3 \) or \( T_W \), i.e., when the processor is not sampling the input, will not cause CPU malfunction.

### 3.1.10 BUS PERFORMANCE ISSUES

Bus cycles occur sequentially, but do not necessarily come immediately one after another, that is the bus may remain idle for several \( T \) states between each bus access initiated by the 80186. The reader should recall that a separate unit, the bus interface unit, fetches opcodes from memory, while the execution unit actually executes the pre-fetched instructions. The number of clock cycles required to execute an 80186 instruction vary from 2 clock cycles for a register to register move to 67 clock cycles for an integer divide.

If a program contains many long instructions, program execution will be CPU limited, that is, the instruction queue will be constantly filled. Thus, the execution unit does not need to wait for an instruction to be fetched. If a program contains mainly short instructions (for example, data move instructions), the execution will be bus limited. Here, the execution unit will have to wait often for an instruction to be fetched before it continues. Programs illustrating this effect and performance degradation of each with the addition of wait states are given in appendix G.

Although the amount of bus utilization will vary considerably from one program to another, a typical instruction mix on the 80186 will require greater bus utilization than the 8086. The 80186 executes most instructions in fewer clock cycles, thus requiring instructions from the queue at a faster rate. This also means that the effect of wait states is more pronounced in an 80186 system than in an 8086 system. In all but a few cases, however, the performance degradation incurred by adding a wait state is less than might be expected because instruction fetching and execution are performed by separate units.

### 3.2 Example Memory Systems

#### 3.2.1 2764 INTERFACE

With the above knowledge of the 80186 bus, various memory interfaces may be generated. One of the simplest is the example EPROM interface shown in Figure 20.

![Figure 19. Valid SRDY Transitions](image-url)
The addresses are latched using the circuit shown earlier. Note that the A0 line of each EPROM is connected to the A1 address line from the 80186, not the A0 line. Remember, A0 only signals a data transfer on the lower 8 bits of the 16-bit data bus. The EPROM outputs are connected directly to the address/data inputs of the 80186, and the 80186 RD signal is used as the OE for the EPROMs.

The chip enable of the EPROM is driven directly by the chip select output of the 80186 (see section 8). In this configuration, the access time calculation for the EPROMs are:

\[
\text{time from address: } (3 + N) \cdot t_{CLCL} - t_{CLAV} - t_{PD} (\text{latch}) - t_{DVCL}
\]

\[
\text{time from chip select: } (3 + N) \cdot t_{CLCL} - t_{CLCSV} - t_{DVCL}
\]

\[
\text{time from RD (OE): } (2 + N) \cdot t_{CLCL} - t_{CLRL} - t_{DVCL}
\]

where:

\[t_{CLAV} = \text{time from clock low in } T_1 \text{ until addresses are valid}\]

\[t_{CLCL} = \text{clock period of processor}\]

\[t_{PD} = \text{time from input valid of latch until output valid of latch}\]

\[t_{DVCL} = 186 \text{ data valid input setup time until clock low time in } T_4\]

\[t_{CLCSV} = \text{time from clock low in } T_1 \text{ until chip selects are valid}\]

\[t_{CLRL} = \text{time from clock low in } T_2 \text{ until RD goes low}\]

\[N = \text{number of wait states inserted}\]

The only significant parameter not included above is \(t_{RHAV}\), the time from RD inactive (high) until the 80186 begins driving address information. The output float time of the EPROM must be within this spec. If slower EPROMs are used, a discrete buffer must be inserted between the EPROM data lines and the address/data bus, since these devices may continue to drive data information on the multiplexed address/data bus when the 80186 begins to drive address information for the next bus cycle.

### 3.2.2 8203 DRAM INTERFACE

An example 8203/DRAM interface is shown in Figure 21. The 8203 provides all required DRAM control signals, address multiplexing, and refresh generation. In this circuit, the 8203 is configured to interface to 64K DRAMs.

All 8203 cycles are generated off control signals (RD and WR) provided by the 80186. These signals will not go active until \(T_2\) of the bus cycle. In addition, since the 8203 clock (generated by the internal crystal oscillator of the 8203) is asynchronous to the 80186 clock, all memory requests by the 80186 must be synchronized to the 8203 before the cycle will be run. To minimize this synchronization time, the 8203 should be used with the highest speed crystal that will maintain DRAM compatibility. If a 25 MHz crystal is used (the maximum allowed by the 8203) two wait states will be required by the example circuit when using 150 ns DRAMs with an 8 MHz 80186, and three wait states if 200 ns DRAMs are used (see Figure 22).

The entire DRAM array controlled by the 8203 can be selected by one or a group of the 80186 provided chip selects. These chip selects can also insert the wait states required by the interface.
Since the 8203 is operating asynchronously to the 80186, the RDY output of the 8203 must be synchronized to the 80186. The 80186 ARDY line provides the necessary ready synchronization. The 8203 ready outputs operate in a normally not ready mode, that is, they are only driven active when an 8203 cycle is being executed, and a refresh cycle is not being run. The 8203 SACK is presented to the 80186 only when the DRAM is being accessed. Notice that the SACK output of the 8203 is used, rather than XACK. Since the 80186 will insert at least one full CPU clock cycle between the time RDY is sampled active and the time data must be present on the data bus, the XACK signal would insert unnecessary additional wait states, since it does not indicate ready until valid data is available from the memory.

### 3.2.3 8207 DRAM INTERFACE

The 8207 advanced dual-port DRAM controller provides a high performance DRAM memory interface specifically for 80186 microcomputer systems. This controller provides all address multiplexing and DRAM refresh circuitry. In addition, it synchronizes and arbitrates memory requests from two different ports (e.g., an 80186 and a Multibus), allowing the two ports to share memory. Finally, the 8207 provides a simple interface to the 8206 error detection and correction chip.

The simplest 8207 (and also the highest performance) interface is shown in Figure 23. This shows the 80186 connected to an 8207 using the 8207 slow cycle, synchronous status interface. In this mode, the 8207 decodes the cycle to be run directly from the status lines of the 80186. In addition, since the 8207 CLOCKIN is driven by the CLKOUT of the 80186, any performance degradation caused by required memory request synchronization between the 80186 and the 8207 is not present. Finally, the entire memory array driven by the 8207 may be selected using one or a group of the 80186 memory chip selects, as in the 8203 interface above.

The 8207 AACK signal generates a synchronous ready signal to the 80186 in the above interface. Since dynamic memory periodically requires refreshing, 80186 ac-
NOTES:
1. tCLEL: Clock low until read low max
2. tCR: Command active until RAS max
3. tCC: Command active until CAS max
4. tCAC: Access time from CAS max
5. tISOU: Input to output delay max
6. tOVCL: Data valid to clock low (data in set up) min
Total Access Time = tCLEL + tCC + tCAC + tISOU + tOVCL

3.3 HOLD/HLDA Interface

The 80186 employs a HOLD/HLDA bus exchange protocol. This protocol allows other asynchronous bus masters (i.e., ones which drive address, data, and control information on the bus) to gain control of the bus.

3.3.1 HOLD RESPONSE

In the HOLD/HLDA protocol, a device requiring bus control (e.g., an external DMA device) raises the HOLD line. In response to this HOLD request, the 80186 will raise its HLDA line after it has finished its current bus activity. When the external device is finished with the bus, it drops its bus HOLD request. The 80186 responds by dropping its HLDA line and resuming bus operation.

When the 80186 recognizes a bus hold by driving HLDA high, it will float many of its signals (see Figure 24). AD0–AD15 and DEN are floated within...
tCLAZ after the same clock edge that HLDA is driven active. A16–A19, RD, WR, BHE, DT/R, and S0–S2 are floated within tCHCZ after the clock edge immediately before the clock edge on which HLDA comes active. AI6–AI9, RD, WR, BHE, DT/R, and SO–S2 are floated within tCHCZ after the clock edge immediately before the clock edge on which HLDA comes active.

Only the above mentioned signals are floated during bus HOLD. Of the signals not floated by the 80186, some have to do with peripheral functionality (e.g., TMR OUT). Many others either directly or indirectly control bus devices. These signals are ALE and all the chip select lines (UCS, LCS, MCS0–3, and PCS0–6).

3.3.2 HOLD/HLDA TIMING AND BUS LATENCY

The time required between HOLD going active and the 80186 driving HLDA active is known as bus latency. Many factors affect bus latency, including synchronization delays, bus cycle times, locked transfer times and interrupt acknowledge cycles.

The HOLD request line is internally synchronized by the 80186, and may therefore be an asynchronous input. To guarantee recognition on a particular clock edge, it must satisfy setup and hold times to the falling edge of the CPU clock. A full CPU clock cycle is required for synchronization (see Appendix B). If the bus is idle, HLDA will follow HOLD by two CPU clock cycles plus a small amount of setup and propagation delay time. The first clock cycle synchronizes the input; the second signals the internal circuitry to initiate a bus hold (see Figure 25).

Many factors influence the number of clock cycles between a HOLD request and a HLDA. These make bus latency longer than the best case shown above. Perhaps the most important factor is that the 80186 will not relinquish the local bus until the bus is idle. The bus can become idle only at the end of a bus cycle. The 80186 will normally insert no T1 states between T4 and T1 of the next bus cycle if it requires any bus activity (e.g., instruction fetches or I/O reads). However, the 80186 may not have an immediate need for the bus after a bus cycle, and will insert T1 states independent of the HOLD input (see Section 3.1.1).

When the HOLD request is active, the 80186 will be forced to proceed from T4 to T1 in order that the bus may be relinquished. HOLD must go active 3 T-states before the end of a bus cycle to force the 80186 to insert idle T-states after T4 (one to synchronize the request, and one to signal the 80186 that T4 of the bus cycle will be followed by idle T-states, see section 3.1.1). After the bus cycle has ended, the bus hold will be immediately acknowledged. If, however, the 80186 has already determined that an idle T-state will follow T4 of the current bus cycle, HOLD need go active only 2 T-states before the end of a bus cycle to force the 80186 to relinquish the bus. This is because the external HOLD request is not required to force the generation of idle T-states. Figure 26 graphically portrays the scenarios depicted above.
An external HOLD has higher priority than both the 80186 CPU or integrated DMA unit. However, an external HOLD will not separate the two cycles needed to perform a word access when the word accessed is located at an odd location (see Section 3.1.3). In addition, an external HOLD will not separate the two-to-four bus cycles required for the integrated DMA unit to perform a transfer. Each of these factors will add to the bus latency of the 80186.

Another factor influencing bus latency time is locked transfers. Whenever a locked transfer is occurring, the 80186 will not recognize external HOLDS (nor will it recognize internal DMA bus requests). Locked transfers are programmed by preceding an instruction with the LOCK prefix. String instructions may be locked. Since string transfers may require thousands of bus cycles, bus latency time will suffer if they are locked.

The final factor affecting bus latency time is interrupt acknowledge cycles. When an external interrupt controller is used, or if the integrated interrupt controller is used in Slave mode (see Section 4.4.1) the 80186 will run two interrupt acknowledge cycles back to back.
These cycles are automatically “locked” and will never be separated by bus HOLD. See Section 6.5 on interrupt acknowledge timing for more information concerning interrupt acknowledge timing.

3.3.3 COMING OUT OF HOLD

When the HOLD input goes inactive, the processor lowers its HLDA line in a single clock as shown in Figure 27. If there is pending bus activity, only two \( T_1 \) states will be inserted after HLDA goes inactive and status information will go active during the last idle state concerning the bus cycle about to be run (see Section 3.1.1). If there are no bus cycles to be run by the CPU, it will continue to float all lines until the last \( T_1 \) before it begins its first bus cycle after the HOLD.

A special mechanism exists on the 80C186/80C188 to provide for DRAM refreshing while the bus is in HOLD. If the refresh control unit issues a request to the integrated bus controller while HOLD is in effect, the processor lowers HLDA. It is the responsibility of the external bus master to release the bus by deasserting HOLD so that the refresh cycle can take place (see Figure 28). The external master can then reassume control of the bus subject to the usual requirements placed on the HOLD input.

![Figure 27. 80186 Coming out of Hold](image1)

![Figure 28. Release of 80C186/80C188 HOLD to Run Refresh Cycle](image2)
3.4 Differences between the 8086 Bus and the 80186 Bus

The 80186 bus was defined to be upward compatible with the 8086 bus. As a result, the 8086 bus interface components (the 82C88 bus controller and the 8289 bus arbiter) may be used with the 80186. There are a few significant differences between the two processors which should be considered.

CPU Duty Cycle and Clock Generator

The 80186 employs an integrated clock generator which provides a 50% duty cycle CPU clock. This is different from the 8086, which utilizes an external clock generator to provide 33% ('I. high, % low) CPU clock. The following points relate to 80186 clock generation:

1) The 80186 uses a crystal or external frequency input twice the desired processor clock frequency.
2) No oscillator output is available from the 80186 internal oscillator.
3) The 80186 does not provide a clock output at reduced frequency from the 80186. However, a timer output may be easily programmed for this purpose.
4) Interfacing the 80186 to devices needing a 33% duty cycle clock (for example, the 8087) is possible, but requires careful timing analysis.
5) Care should be exercised not to exceed the drive capability of the 80186 CLKOUT pin.

Local Bus Controller and Control Signals

The 80186 simultaneously provides both local bus controller outputs and status outputs for use with the 82C88 bus controller. This is different from the 8086 where the local bus controller outputs are sacrificed if status outputs are desired. These differences will manifest themselves in 8086 systems and 80186 systems as follows:

1) Because the 80186 can simultaneously provide local bus control signals and status outputs, many systems supporting both a system bus (e.g., a MULTIBUS®) and a local bus will not require two separate external bus controllers, that is, the 80186 bus control signals may be used to control the local bus while the 80186 status signals are concurrently connected to the 82C88 bus controller to drive the control signals of the system bus.
2) The ALE signal of the 80186 goes active a clock phase earlier on the 80186 then on the 8086 or 82C88. This minimizes address propagation time through the address latches, since typically the delay time through these latches from inputs valid is less than the propagation delay from the strobe input active.
3) The 80186 RD input must be tied low to provide queue status outputs from the 80186 (see Figure 29). When so speeded into “queue status mode,” the ALE and WR outputs provide queue status information. Notice that queue status information is available one clock phase earlier from the 80186 than from the 8086 (see Figure 30).
HOLD/HLDA vs. RQ/GT

As discussed earlier, the 80186 uses a HOLD/HLDA protocol for exchanging bus mastership (like the 8086 in min mode) rather than the RQ/GT protocol used by the 8086 in max mode. This allows compatibility with Intel's bus master peripheral devices (for example the 82586 Ethernet controller or 82730 high performance CRT controller/text coprocessor).

Status Information

The 80186 does not provide S3–S5 status information. On the 8086, S3 and S4 provide information regarding the segment register generating the physical address of the current bus cycle. S5 provides information concerning the state of the interrupt enable flip-flop. These status lines are always low on the 80186.

Status signal S6 indicates whether the current bus cycle is initiated by either the CPU or a DMA device. Subsequently, it is always low on the 8086. On the 80186, it is low whenever the current bus cycle is initiated by the 80186 CPU, and is high when the current bus cycle is initiated by the integrated DMA unit.

Miscellaneous

The 80186 does not provide early and late write signals, as does the 82C88 bus controller. The WR signal generated by the 80186 corresponds to the early write signal of the 82C88. This means that data is not stable on the address/data bus when this signal is driven active.

The 80186 also does not provide both I/O and memory read and write command signals. If these signals are desired, an external 82C88 bus controller may be used, or the S2 signal may be used to synthesize both commands (see Section 3.1.6.1).

4.0 DMA UNIT INTERFACING

The 80186 includes a DMA unit consisting of two independent DMA channels. These channels operate independently of the CPU, and drive all integrated bus interface components (bus controller, chip selects, etc.) exactly as the CPU (see Figure 31). This means that bus cycles initiated by the DMA unit are the same as bus cycles initiated by the CPU (except that S6 = 1 during all DMA initiated cycles). Interfacing the DMA unit itself is very simple, since except for the addition of the DMA request connection, it is exactly the same as interfacing to the CPU.

Figure 31. 80186 CPU/DMA Channel Internal Model
4.1 DMA Features

Each of the two DMA channels provides the following features:

- Independent 20-bit source and destination pointers which access the I/O or memory location from which data will be fetched or to which data will be deposited
- Programmable auto-increment, auto-decrement or neither of the source and destination pointers after each DMA transfer
- Programmable termination of DMA activity after a certain number of DMA transfers
- Programmable CPU interruption at DMA termination
- Byte or word DMA transfers to or from even or odd memory or I/O addresses
- Programmable generation of DMA requests by:
  1) the source of the data
  2) the destination of the data
  3) timer 2 (see Section 5)
  4) the DMA unit itself (continuous DMA requests)

4.2 DMA Unit Programming

Each of the two DMA channels contains a number of registers to control channel operation. These registers are included in the 80186 integrated peripheral control block (see Appendix A). These registers include the source and destination pointer registers, the transfer count register and the control register. The layout of the bits in these registers is given in Figures 32 and 33.

The 20-bit source and destination pointers access the complete 1 Mbyte address space of the 80186 and all 20 bits are affected by the auto-increment or auto-decrement unit of the DMA. The address space is seen as a flat, linear array without segments. Even though the usual I/O addressability of the 80186 is 64 Kbytes, it is possible to perform I/O accesses over a 1 Mbyte address range. Therefore, it is important to program the upper four bits of the pointer registers to 0 if routine I/O addresses are desired.

After every DMA transfer the 16-bit DMA transfer count register it is decremented by 1, whether a byte transfer or a word transfer has occurred. If the TC bit in the DMA control register is set, the DMA ST/STOP bit (see below) will be cleared when this register goes to 0, causing all DMA activity to cease. A transfer count of zero allows 65536 \((2^{16})\) transfers.

Upon reset, the contents of the DMA pointer registers and transfer count registers are indeterminate; initialization of all the bits should be practiced.

The DMA control register (see Figure 33) contains bits which control various channel characteristics, including for each of the data source and destination whether the pointer points to memory or I/O space, or whether the pointer will be incremented, decremented or left alone after each DMA transfer. It also contains a bit which selects byte or word transfers. Two synchronization bits determine the source of the DMA requests (see Section 4.7). The TC bit determines whether DMA activity will cease after a programmed number of DMA transfers, and the INT bit enables interrupts to the processor when this has occurred (note that an interrupt will not be generated to the CPU when the transfer count register reaches zero unless both the INT bit and the TC bit are set).
The control register also contains a start/stop (ST/STOP) bit which enables DMA transfers. Whenever this bit is set, the channel is armed, that is, a DMA transfer will occur whenever a DMA request is made to the channel. A companion bit, the CHG/NOCHG bit, allows the DMA control register to be changed without modifying the state of the start/stop bit. The ST/STOP bit will only be modified if the CHG/NOCHG bit is also set during the write to the DMA control register. The CHG/NOCHG bit is write only. It will always be read back as a 0. Because DMA transfers could occur immediately after the ST/STOP bit is set, it should only be set after all other DMA controller registers have been programmed. This bit is automatically cleared when the transfer count register reaches zero and the TC bit in the DMA control register is set, or when the transfer count register reaches zero and unsynchronized DMA transfers are programmed.

All DMA unit programming registers are directly accessible by the CPU. This means the CPU can, for example, modify the DMA source pointer register after 137 DMA transfers have occurred, and have the new pointer value used for the 138th DMA transfer. If more than one register in the DMA channel is being modified at any time that a DMA request may be generated and the DMA channel is enabled (the ST/STOP bit in the control register is set), the register programming values should be placed in memory locations and moved into the DMA registers using a locked string move instruction. This will prevent a DMA transfer from occurring after only some of the register values have changed. The above also holds true if a read/modify/write type of operation is being performed (e.g., ANDing off bits in a pointer register in a single AND instruction to a pointer register mapped into memory space).
4.3 DMA Transfers

Every 80186 DMA transfer consists of two independent bus cycles, a fetch cycle and a deposit cycle (see Figure 34). During the fetch cycle, the byte or word data is accessed according to the source pointer register. The data is read into an internal temporary register which is not accessible by the CPU. During the deposit cycle, the data is written to memory or I/O space at the address in the destination pointer register. These two bus cycles cannot be separated by a bus HOLD, a refresh cycle, or any other condition except RESET. DMA bus cycles are identical to bus cycles initiated by the CPU except that the S6 status line is driven to a logic one state.

4.4 DMA Requests

Each DMA channel has a single DMA request line by which an external device may request a DMA transfer. The synchronization bits in the DMA control register determine whether this line is interpreted to be connected to the source or destination of the DMA data. All transfer requests on this line are synchronized internally to the CPU clock before being presented to internal DMA logic. In addition to external requests, DMA requests may be generated whenever the internal Timer 2 times out, or continuously by programming the synchronization bits in the DMA control register for unsynchronized DMA transfers.

4.4.1 DMA REQUEST TIMING AND LATENCY

Before any DMA request can be generated, the 80186 internal bus must be granted to the DMA unit. A certain amount of time is required for the CPU to grant this internal bus to the DMA unit. The time between a DMA request being issued and the DMA transfer being run is known as DMA latency. Many of the issues concerning DMA latency are the same as those concerning bus latency (see Section 3.3.2). The only important difference is that external HOLD and refresh always have bus priority over an internal DMA transfer. Thus, the latency time of an internal DMA cycle will suffer during an external bus HOLD.

Each DMA channel has a programmed priority relative to the other DMA channel. Both channels may be programmed to be the same priority, or one may be programmed to be of higher priority than the other channel. If both channels are active, DMA latency will suffer on the lower priority channel. If both channels are active and both channels are of the same programmed priority, DMA transfer cycles will alternate between the two channels (i.e., the first channel will perform a fetch and deposit, followed by a fetch and deposit by the second channel, etc.).

Figure 34. Example DMA Transfer Cycle on the 80186

NOTES:
1. Source address
2. Source data
3. Destination address
4. Destination data
5. If a source or destination address overlaps an active chip select region, the chip select will go active.
6. Wait states are inserted by programming the chip select/ready logic for an active address region, or by the external ready pins.
The minimum timing required to generate a DMA cycle is shown in Figure 35. Note that the minimum time from DRQ becoming active until the beginning of the first DMA cycle is 4 CPU clock cycles. This time is independent of the number of wait states inserted in the bus cycle. The maximum DMA latency is a function of other processor activity.

Also notice that if DRQ is sampled active at 1 in Figure 35, the DMA cycle will be executed, even if the DMA request goes inactive before the beginning of the first DMA cycle. This does not mean that the DMA request is latched into the processor. Quite the contrary, DRQ must be active at a certain time before the end of a bus cycle for the request to be recognized by the processor. If DRQ goes inactive before that window, then no DMA cycles will be run.

4.5 DMA Acknowledge

The 80186 generates no explicit DMA acknowledge (DACK) signal. Instead, the 80186 performs a read or write directly to the DMA requesting device. If required, a DMA acknowledge signal can be generated by a decode of an address, or by merely using one of the PCS lines (see Figure 36). Note ALE must be used to factor DACK because addresses are not guaranteed stable when chip selects go active.

---

**Figure 35. DMA Request Timing on the 80186 (showing minimum response time to request)**

**Figure 36. DMA Acknowledge Synthesis from the 80186**

NOTE:
1. A6 is arbitrary and is only used to differentiate DMA and non-DMA bus cycles
4.6 Internally Generated DMA Requests

DMA transfer requests may originate from two of the integrated peripherals in the 80186. The source may be either the DMA control unit or Timer 2.

The DMA channel can be programmed so that whenever Timer 2 reaches its maximum count, a DMA request will be generated. This feature is selected by setting the TDRQ bit in the DMA channel control register. A DMA request generated in this manner will be latched in the DMA controller, so that once the timer request has been generated, it cannot be cleared except by running the DMA cycle or by clearing the TDRQ bits in both DMA control registers. Before any DMA requests are generated in this mode, Timer 2 must be initialized and enabled.

A timer requested DMA cycle being run by either DMA channel will reset the timer request. Thus, if both channels are using it to request a DMA cycle, only one DMA channel will execute a transfer for every timeout of Timer 2. Another implication of having a single bit timer DMA request latch in the DMA controller is that if another Timer 2 timeout occurs before a DMA channel has a chance to run a DMA transfer, the first request will be lost.

The DMA channel can also be programmed to provide its own DMA requests. In this mode, DMA transfer cycles will be run continuously at the maximum bus bandwidth until the preprogrammed number of DMA transfers have occurred. This mode is selected by programming the synchronization bits in the DMA control register for unsynchronized transfers. Note that in this mode, the DMA controller will monopolize the CPU bus, i.e., the CPU will not be able to perform opcode fetching, memory operations, etc., while the DMA transfers are occurring. Also notice that the DMA will only perform the number of transfers indicated in the maximum count register regardless of the state of the TC bit in the DMA control register.

4.7 Externally Synchronized DMA Transfers

There are two types of externally synchronized DMA transfers. These are source and destination synchronized transfers. These modes are selected by programming the synchronization (SYN) bits in the DMA channel control register. The only difference between the two is the time at which the DMA request pin is sampled to determine if another DMA transfer is immediately required after the currently executing DMA transfer. On source synchronized transfers, this is done such that two transfers may occur one immediately after the other, while on destination synchronized transfers a certain amount of idle time is automatically inserted between two DMA transfers to allow time for the DMA requesting device to drive its DMA request inactive.

4.7.1 SOURCE SYNCHRONIZED DMA TRANSFERS

In a source synchronized DMA transfer, the data source requests the DMA cycle. An example is a floppy disk read from the disk to main memory. In this type of transfer, the device requesting the transfer is read during the fetch cycle of the DMA transfer. Since it takes 4 CPU clock cycles from the time DMA request is sampled to the time the DMA transfer is actually begun, and a bus cycle takes a minimum of 4 clock cycles, the earliest time the DMA request pin will be sampled for another DMA transfer will be at T1 of the deposit cycle of the DMA transfer (assuming no wait states.) This allows 3 or more CPU clock cycles between the time the DMA requesting device receives an acknowledge to its DMA request (around the beginning of T2 of the DMA fetch cycle), and the time it must drive this request inactive (assuming no wait states) to insure that another DMA transfer is not performed if it is not desired (see Figure 37).

4.7.2 DESTINATION SYNCHRONIZED DMA TRANSFERS

In destination synchronized DMA transfers, the data destination requests the DMA transfer. An example of this would be a floppy disk write from main memory to the disk. In this type of transfer, the device requesting the transfer is written during the deposit cycle of the DMA transfer. This causes a problem, since the DMA requesting device will not receive notification of the DMA cycle being run until 3 clock cycles before the end of the DMA transfer (if no wait states are being inserted into the deposit cycle of the DMA transfer) and it takes 4 clock cycles to determine if another DMA cycle should run immediately following the current transfer. To get around this problem, the DMA unit relinquishes the bus after each destination synchronized DMA transfer for at least 2 CPU clock cycles to allow the requesting device time to drop its DMA request if it does not immediately desire another DMA transfer. When the bus is relinquished by the DMA unit, the CPU may resume bus operation. Typically, a CPU initiated bus cycle is inserted between each destination synchronized DMA transfer. If no CPU bus activity is required, however, the DMA unit inserts only 2 CPU clock cycles between the deposit cycle of one DMA transfer and the fetch cycle of the next DMA transfer. This means that the requesting device must drop its request line at least two clock cycles before the end of the deposit cycle regardless of the number of wait states inserted. Figure 37 shows the DMA request going away too late to prevent the immediate generation of another DMA transfer. Any wait states inserted
NOTE:
1. Current DMA source synchronized transfer will not be immediately followed by another DMA transfer.

Figure 37. Source & Destination Synchronized DMA Request Timing

in the deposit cycle of the transfer lengthen the amount of time from the beginning of the deposit cycle to the time DRQ is sampled for another DMA transfer. Thus, if the amount of time a device requires to drop its request after receiving an acknowledge from the 80186 is longer than the 0 wait state 80186 maximum (about 1 clock), wait states can be inserted into the DMA cycle to lengthen the amount of time the device has to drop its request after receiving the DMA acknowledge.

4.8 DMA Halt and NMI

Whenever a Non-Maskable Interrupt is received by the 80186, all DMA activity will be suspended at the end of the current DMA transfer. This is performed by the NMI automatically setting the DMA Halt (DHLT) bit in the interrupt controller status register (see Section 6.3.7). The timing of NMI required to prevent a DMA cycle from occurring is shown in Figure 38. After the NMI has been serviced, the DHLT bit can be cleared by the programmer to resume DMA activity (i.e., it is not automatically cleared when entering the NMI service routine). The DHLT bit is automatically cleared when the IRET instruction is executed. In either case, DMA activity resumes exactly as it left off, i.e., none of the DMA control registers are modified. This DHLT bit may also be set by the programmer to prevent DMA activity during critical sections of code. The DHLT bit does not function when the integrated interrupt controller is configured for Slave Mode.

4.9 Example DMA Interfaces

4.9.1 8272 FLOPPY DISK INTERFACE

An example DMA interface to the 8272 Floppy Disk Controller is shown in Figure 39. This shows how a typical DMA device can be interfaced to the 80186. An example floppy disk software driver for this interface is given in Appendix C.

The data lines of the 8272 are connected, through buffers, to the 80186 AD0-AD7 lines. The buffers are required because the 8272 will not float its output drivers quickly enough to prevent contention with the 80186 upon the next bus cycle (see Section 3.1.5).

DMA acknowledge for the 8272 is driven by an address decode within the region assigned to PCS2. If PCS2 is assigned to be active between I/O locations 0500H and 057FH, then an access to I/O location 0500H will enable only the chip select, while an access to I/O location 0501H will enable both the chip select and the DMA acknowledge. Remember, ALE must be factored
NOTES:
1. DMA request synchronization
2. Decision: Will DMA cycle be run?
   Answer: No DMA request is active but DHLT is set
   (from NMI request)
3. NMI synchronization time
4. Logic delay time from synchronized NMI until DHLT set (note: DHLT is in the interrupt control status register)

Figure 38. NMI and DMA Interaction

Figure 39. Example 8272/80186 DMA Interface
into the DACK generation logic because addresses are not guaranteed stable when the chip selects become active.

Notice that the TC line of the 8272 is driven by a very similar circuit as the one generating DACK (except for the reversed sense of the output!). This line is used to terminate an 8272 command before the command has completed execution. Thus, the TC input to the 8272 is software driven in this case. Another method of driving the TC input would be to connect the DACK signal to one of the 80186 timers, and program the timer to output a pulse to the 8272 after a certain number of DMA cycles have been run (see next section for 80186 timer information).

The above discussion assumed that a single 80186 PCS line is free to generate all 8272 select signals. If more than one select is free, however, different 80186 generated PCS lines could be used for each function. For example, PCS2 could be used to select the 8272 and PCS3 could be used to drive the DACK line of the 8272.

DMA requests are delayed by two clock periods in going from the 8272 to the 80186. This is required by the 8272 TRQR (time from DMA request to DMA RD going active) spec. This requires many 80186 CPU clock cycles, well beyond the 5 minimum provided by the 80186 (4 clock cycles to the beginning of the DMA bus cycle, 5 to the beginning of T2 of the DMA bus cycle where RD will go active). The two flip-flops add two complete CPU clock cycles to this response time.

DMA request will go away after DACK is presented to the 8272. During a DMA write cycle (i.e., a destination synchronized transfer), this does not occur soon enough to prevent the immediate generation of another DMA transfer if no wait states are inserted in the deposit cycle to the 8272. Therefore, at least 1 wait state is required by this interface, regardless of the data access parameters of the 8272.

4.9.2 8274 SERIAL COMMUNICATION INTERFACE

An example 8274 synchronous/asynchronous serial chip/80186 DMA interface is shown in Figure 40. The 8274 interface is simpler than the 8272 interface, since it does not require a DMA acknowledge signal, and the 8274 does not require the length of time between a DMA request and the DMA read or write cycle that the 8272 does. An example serial driver using the 8274 in DMA mode with the 80186 is given in Appendix C.

The data lines of the 8274 are connected through buffers to the 80186 AD0–AD7 lines. Again, these are required not because of bus drive problems, but because the 8274 does not float its drivers before the 80186 begins driving address information on its address/data bus. If both the 8274 and the 8272 are included in the same 80186 system, they could share the same data bus buffer (as could any other peripheral devices in the system).

The 8274 does not require a DMA acknowledge signal. The first read or write of the 8274 data register after the 8274 generates the DMA request signal clears the DMA request. The time between the control signal (RD or WR) going active and the 8274 dropping its DMA request during a DMA write requires at least one wait state be inserted into the DMA write cycle.

5.0 TIMER UNIT INTERFACING

The 80186 includes a timer unit which consists of three independent 16-bit timers. These timers operate independently of the CPU. Two have input and output pins allowing counting of external events and generation of arbitrary waveforms. The third can be used as a timer, as a prescaler for the other two timers, or as a DMA request source.
5.1 Timer Operation

The internal timer unit on the 80186 can be modeled by a single counter element, time multiplexed to three register banks, each of which contains different control and count values. These register banks are, in turn, dual ported between the counter element and the 80186 CPU (see Figure 41). Figure 42 shows the timer element sequencing, and the subsequent constraints on input and output signals. There is no connection between the sequencing of the counter element through the timer register banks and the Bus Interface Unit's sequencing through T-states. Timer operation and bus interface operation are completely asynchronous.

5.2 Timer Unit Programming

Each timer is controlled by a block of registers (see Figure 43). Each of these registers can be read or written whether or not the timer is operating. All processor accesses to these registers are synchronized to all counter element accesses to these registers, meaning that one will never read a count register in which only half of the bits have been modified. Because of this synchronization, one wait state is automatically inserted into any access to the timer registers. Unlike the DMA unit, locking accesses to timer registers will not prevent the timer's counter elements from accessing the timer registers.

Each timer has a 16-bit count register which is incremented for each timer event. A timer event can be a low-to-high transition on a TIMERIN pin (for Timers 0 and 1), a pulse generated every fourth CPU clock, or a time out of Timer 2 (for Timers 0 and 1). Because the count register is 16 bits wide, up to 65536 \((2^{16})\) timer events can be counted. Upon RESET, the contents of the count registers are indeterminate and they should be initialized to zero before any timer operation.

Each timer includes a maximum count register. Whenever the timer count register is equal to the maximum count register, the count register resets to zero, so the maximum count value can never be stored in the count register. This maximum count value may be written while the timer is operating. A maximum count value of 0 implies a maximum count of 65536, a maximum count value of 1 implies a maximum count of 1, etc. Only equivalence between the count value and the maximum count register value is checked. This means that the count value will not be cleared if the value in the count register is greater than the value in the maximum count register. This situation only occurs by programmer intervention, either by setting the value in the count register greater than the value in the maximum count register, or by setting the value in the maximum count register to be less than the value in the count register. If the timer is programmed in this way, it will count to the maximum possible count (FFFFH), increment to 0, then count up to the value in the maximum count register. The TC bit in the timer control register will not be set when the counter overflows to 0, nor will an interrupt be generated from the timer unit.

![Figure 41. 80186 Timer Model](210973-48)
NOTES:
1. Timer in 0 resolution time
2. Timer in 1 resolution time
3. Modified count value written into 80186 Timer 0 count register
4. Modified count value written into 80186 Timer 1 count register

Figure 42. 80186 Counter Element Multiplexing and Timer Input Synchronization

Figure 43. 80186 Timer Register Layout
Timers 0 and 1 each contain an additional maximum count register. When both maximum count registers are used, the timer will first count up to the value in maximum count register A, reset to zero, count up to the value in maximum count register B, and reset to zero again. The ALTernate bit in the timer control register determines whether one or both maximum count registers are used. If this bit is low, only maximum count register A is used; maximum count register B is ignored. If it is high, both registers are used. The RIU (register in use) bit in the timer control register indicates which maximum count register is currently being used. This bit is 0 when maximum count register A is being used, 1 when maximum count register B is being used. The RIU bit is read only. It will always be read 0 in single maximum count register mode (since only maximum count register A will be used).

Each timer can generate an interrupt whenever the timer count value reaches a maximum count value. Interrupts result whenever the timer count matches maximum count A (for Timer 2 or Timers 0 and 1 in single max count mode) and whenever the timer count matches maximum count B (for Timers 0 and 1 in dual max count mode). In addition, the MC (maximum count) bit in the timer control register is set whenever the timer count reaches a maximum count value. This bit is never automatically cleared, i.e., programmer intervention is required. If a timer generates a second interrupt request before the first interrupt request has been serviced, the first interrupt request to the CPU will be lost.

Each timer has an ENable bit in the timer control register. The timer will count timer events only when this bit is set. Any write to the timer control register will modify the ENable bit only if the INHibit bit is also set. The INHibit bit in the timer control register allows selective updating of the timer ENable bit. The value of the INHibit bit is not stored in a write to the timer control register; it will always be read as a logic zero.

Each timer has a CONTinuous bit in the timer control register. If this bit is cleared, the timer ENable bit will be automatically cleared at the end of each timing cycle. If a single maximum count register is used, the end of a timing cycle occurs when the count value resets to zero after reaching the value in maximum count register A. If dual maximum count registers are used, the end of a timing cycle occurs when the count value resets to zero after reaching the value in maximum count register B. If the CONTinuous bit is set, the ENable bit will never be automatically reset. Thus, after each timing cycle, another timing cycle will automatically begin. For example, in single maximum count register mode, the timer will count up to the value in maximum count register A, reset to zero, ad infinitum. In dual maximum count register mode, the timer will count up the value in maximum count register A, reset to zero, count up to the value in maximum count register B, reset to zero, count up to the value in maximum count register A, reset to zero, et cetera.

### 5.3 Timer Events

Each timer counts events. All timers can use a transition of the CPU clock as an event. Because of the counter element multiplexing, the timer count value will be incremented every fourth CPU clock. For Timer 2, this is the only timer event which can be used. For Timers 0 and 1, this event is selected by clearing the EXTernal and Prescaler bits in the timer control register.

Timers 0 and 1 can use Timer 2 reaching its maximum count as a timer event. This is selected by clearing the EXTernal bit and setting the Prescaler bit in the timer control register. When this is done, the timer will increment whenever Timer 2 resets to zero having reached its own maximum count. Note that Timer 2 must be initialized and running in order to increment the value in the other timer/counter.

Timers 0 and 1 can also be programmed to count low-to-high transitions on the external input pin. Each transition on the external pin is synchronized to the 80186 clock before it is presented to the timer circuitry, (see Appendix B for information on 80186 synchronizers). The timer counts transitions on the input pin: the input value must go low, then go high to cause the timer increment. Transitions on this line are latched. Because of the counter element multiplexing, the maximum rate at which the timer can count is 1/4 of the CPU clock rate.

### 5.4 Timer Input Pin Operation

Timers 0 and 1 each have individual timer input pins. All low-to-high transitions on these input pins are synchronized, latched, and presented to the counter element when the particular timer is being serviced by the counter element.

Signals on this input can affect timer operation in three different ways. The manner in which the pin signals are used is determined by the EXTernal and RTG (retrigger) bits in the timer control register. If the EXTernal bit is set, transitions on the input pin will cause the timer count value to increment if the timer is enabled (the ENable bit in the timer control register is set). Thus, the timer counts external events. If the EXTernal bit is cleared, all timer increments are caused by either the CPU clock or by Timer 2 timing out. In this mode, the RTG bit determines whether the input pin will enable timer operation, or whether it will retrigger timer operation.
When the EXTernal bit is low and the RTG bit is also low, the timer will count internal timer events only when the timer input pin is high and the ENable bit in the timer control register is set. Note that in this mode, the pin is level sensitive, not edge sensitive. A low-to-high transition on the timer input pin is not required to enable timer operation. If the pin is tied high, the timer will be continually enabled. The timer enable input signal is completely independent of the ENable bit in the timer control register; both must be high for the timer to count. Example uses for the timer in this mode would be a real time clock or a baud rate generator.

When the EXTernal bit is low and the RTG bit is high, every low-to-high transition on the timer input pin causes the timer count register to reset to zero. This mode of operation can be used to generate a retriggerable digital one-shot. After the timer is enabled (i.e., the ENable bit in the timer control register is set), timer operation (counting) will begin only after the first low-to-high transition of the timer input pin has been detected. If another low-to-high transition occurs on the input pin before the end of the timer cycle, the timer will reset to zero and begin the timer cycle again. A timer cycle is defined as the time the timer is counting from 0 to the maximum count (either max count A or max count B). This means that in the dual max count mode, the RIU bit is not set if the timer is reset by the low-to-high transition on the input pin. Should a timer reset occur when RIU is set (indicating max count B), the timer will again begin to count up to max count B before resetting the RIU bit. Thus, when the ALTerminate bit is set, a timer reset will retrigger (or extend) the duration of the current max count in use (which means that either the low or high level of the timer output will be extended). If the CONTinuous bit in the timer control register is cleared, the timer ENable bit will automatically be cleared whenever a timer cycle has been completed (max count is reached). If the CONTinuous bit in the timer control register is set, the timer will reset to zero and begin another timer cycle whenever the current cycle has completed.

5.5 Timer Output Pin Operation

Timers 0 and 1 each have a timer output pin which can perform two functions at programmer option. The first is a single pulse indicating the end of a timing cycle. The second is a level indicating the maximum count register currently being used. The timer outputs operate as outlined below whether internal or external clocking of the timer is used. If external clocking is used, however, the user should remember that the time between an external transition on the timer input pin and the time this transition is reflected in the timer output pin will vary depending on when the input transition occurs relative to the timer being serviced by the counter element.

When the timer is in single maximum count register mode, the timer output pin will go low for a single CPU clock one clock after the timer is serviced by the counter element where maximum count is reached (see Figure 44). This mode is useful when using the timer as a baud rate generator.

When the timer is programmed in dual maximum count register mode, the timer output pin indicates which maximum count register is being used. It is low if maximum count register B is being used and high if maximum count register A is being used. If the timer is programmed in continuous mode (the CONTinuous bit in the timer control register is set), this pin could generate a waveform of almost any duty cycle. For example, if maximum count register A contained 10 and maximum count register B contained 20, a 33% duty cycle waveform would be generated.

![Figure 44. 80186 Timer Out Signal](image-url)
5.6 Sample 80186 Timer Applications

The 80186 timers can be substituted in almost any application for a discrete timer circuit. Such applications include baud rate generation, digital one-shots, pulse width modulation, event counters and pulse width measurement.

5.6.1 80186 TIMER REAL TIME CLOCK

The sample program in appendix D shows the 80186 timer being used with the 80186 CPU to form a real time clock. In this implementation, Timer 2 is programmed to provide an interrupt to the CPU every millisecond. The CPU then increments memory based clock variables.

5.6.2 80186 TIMER BAUD RATE GENERATOR

The 80186 timers can also be used as baud rate generators for serial communication controllers (e.g., the 8274). Figure 46 shows this simple connection, and the code to program the timer as a baud rate generator is included in Appendix D.

5.6.3 80186 TIMER EVENT COUNTER

The 80186 timer can be used to count events. Figure 47 shows a hypothetical set up in which the 80186 timer will count the interruptions in a light source. The number of interruptions can be read directly from the count register of the timer, since the timer counts up, i.e., each interruption in the light source will cause the timer count value to increase. The code to set up the 80186 timer in this mode is included in Appendix D.
6.0 80186 INTERRUPT CONTROLLER INTERFACING

The tasks performed by the 80186 integrated interrupt controller include synchronization of interrupt requests, prioritization of interrupt requests, and request type vectoring in response to a CPU interrupt acknowledge. It can be a master to two external 8259A interrupt controllers or can be a slave to an external master interrupt controller.

6.1 Interrupt Controller Model

The integrated interrupt controller block diagram is shown in Figure 48. It contains registers and a control element. Four inputs are provided for external interfacing to the interrupt controller. Their functions change according to the mode of the interrupt controller. Like the other 80186 integrated peripheral registers, the interrupt controller registers are available for CPU reading or writing at any time.

6.2 Interrupt Controller Operation

The interrupt controller operates in two major modes, Master Mode and Slave Mode. In Master Mode the integrated controller acts as the master interrupt controller for the system, while in Slave Mode the controller operates as a slave to an external master interrupt controller. Some of the interrupt controller registers and interrupt controller pins change definition between these two modes. The difference is when in Master Mode, the interrupt controller presents its interrupt input directly to the 80186 CPU, while in Slave Mode the interrupt controller presents an interrupt output to an external controller (which then presents its interrupt input to the 80186 CPU). Placing the interrupt controller in Slave Mode is done by setting the SLAVE/MASTER bit in the peripheral control block pointer (see Appendix A).

6.3 Interrupt Controller Unit Programming

The interrupt controller has a number of registers which control its operation (see Figure 49). Some of these change their function between the two major modes of the interrupt controller. The differences are indicated in the following section. If not indicated, the function and implementation of the registers is the same in the two modes of operation. The interaction among the various interrupt controller registers is shown in the flowcharts in Figures 57 and 58.

6.3.1 CONTROL REGISTERS

Each source of interrupt to the 80186 has a control register in the internal controller. These registers

---

Figure 48. 80186 Interrupt Controller Block Diagram
contain three bits which select one of eight interrupt priority levels for the device (0 is highest priority, 7 is lowest priority), and a mask bit to enable the interrupt (see Figure 50). When the mask bit is zero, the interrupt is enabled, when it is one, the interrupt is masked.

There are seven control registers in the 80186 integrated interrupt controller. In Master Mode, four of these serve the external interrupt inputs, one each for the two DMA channels, and one for the collective timer interrupts. In Slave Mode, the external interrupt inputs are not used, so each timer has its own individual control register.

### 6.3.2 REQUEST REGISTER

The interrupt controller includes an interrupt request register (see Figure 51). This register contains seven active bits, one for every interrupt source with an interrupt control register. Whenever an interrupt request is made, the bit in the interrupt request register is set regardless of whether the interrupt is enabled. These interrupt request bits are automatically cleared when the interrupt is acknowledged. The D1 and D0 bits of the request register can also be set (requesting a DMA interrupt), or cleared (removing a DMA interrupt request) by programming.

### 6.3.3 MASK REGISTER AND PRIORITY MASK REGISTER

The interrupt controller mask register (see Figure 51) contains a mask bit for each interrupt source associated with an interrupt control register. The bit for an interrupt source in the mask register is the same bit as provided in the interrupt control register; modifying a mask bit in the control register will also modify it in the mask register, and vice versa.

The interrupt priority mask register (see Figure 52) contains three bits which indicate the lowest priority an interrupt may have that will cause an interrupt request to actually be serviced. Interrupts received which have a lower priority will be masked. Upon reset this register is set to the lowest priority of 7 to enable interrupts of any priority. This register may be read or written.
6.3.4 IN-SERVICE REGISTER

The interrupt controller contains an in-service register (see Figure 51). A bit in the in-service register is associated with each interrupt control register so that when an interrupt request by the device associated with the control register is acknowledged by the processor (either by the processor running the interrupt acknowledge or by the processor reading the interrupt poll register) the bit is set. The bit is reset when the CPU issues an End Of Interrupt to the interrupt controller. This register may be both read and written, i.e., the CPU may set in-service bits without an interrupt ever occurring, or may reset them without using the EOI function of the interrupt controller.

6.3.5 POLL AND POLL STATUS REGISTERS

The interrupt controller contains both a poll register and a poll status register (see Figure 53). These registers contain the same information. They have a single bit to indicate an interrupt is pending. This bit is set if an interrupt of sufficient priority has been received. It is automatically cleared when the interrupt is acknowledged. If an interrupt is pending, the remaining bits contain information about the highest priority pending interrupt. These registers are read-only.

Reading the poll register will acknowledge the pending interrupt to the interrupt controller just as if the processor had acknowledged the interrupt through interrupt acknowledge cycles. The processor will not actually run any interrupt acknowledge cycles, and will not vector through a location in the interrupt vector table. The contents of the interrupt request, in-service, poll, and poll status registers will change appropriately. Reading the poll status register will merely transmit the status of the polling bits without modifying any of the other interrupt controller registers.
The poll and poll status registers are not supported in Slave Mode. The state of the bits in these registers in Slave Mode is not defined.

6.3.6 END OF INTERRUPT REGISTER

The interrupt controller contains an End Of Interrupt register (see Figure 54). The programmer issues an End Of Interrupt (EOI) to the controller by writing to this register. After receiving the EOI, the interrupt controller automatically resets the in-service bit for the interrupt. The value of the word written to this register determines whether the EOI is specific or non-specific. A non-specific EOI is specified by setting the non-specific bit in the word written to the EOI register. In a non-specific EOI, the in-service bit of the highest priority interrupt set is automatically cleared, while a specific EOI allows the in-service bit cleared to be explicitly specified. If the highest priority interrupt is reset, the poll and poll status registers change to reflect the next lowest priority interrupt to be serviced. If a less than highest priority interrupt in-service bit is reset, the priority poll and poll status registers will not be modified (because the highest priority interrupt to be serviced has not changed). Only the specific EOI is supported in Slave Mode. This register is write only.

6.3.7 INTERRUPT STATUS REGISTER

The interrupt controller also contains an interrupt status register (see Figure 55). This register contains four programmable bits. Three bits show which timer is causing an interrupt. This is required because in master mode, the timers share a single interrupt control register. A bit in this register is set to indicate which timer has generated an interrupt. The bit associated with a timer is automatically cleared after the interrupt request for the timer is acknowledged. More than one of these bits may be set at a time. The fourth bit is the DMA halt bit (not implemented in Slave Mode). When set, this bit prevents any DMA activity. It is automatically set whenever a NMI is received by the interrupt controller. It can also be set by the programmer. This bit is automatically cleared whenever the IRET instruction is executed. All implemented bits in the interrupt status register are read/write. Do not perform the write operation when interrupts from the timer/counters are possible; a conflict with internal use of the register may lead to incorrect timer interrupt processing.

6.3.8 INTERRUPT VECTOR REGISTER

In Slave Mode only, the interrupt controller contains an interrupt vector register (see Figure 56). This register specifies the 5 most significant bits of the interrupt type vector placed on the CPU bus in response to an interrupt acknowledgement (the lower 3 significant bits of the interrupt type are determined by the priority level of the device causing the interrupt in Slave Mode).
6.4 Interrupt Sources

The 80186 interrupt controller receives and arbitrates among many different interrupt request sources, both internal and external. Internal interrupts are processed by the interrupt controller in either Master Mode or Slave Mode. External interrupts are processed by the integrated interrupt controller only in Master Mode. Each interrupt source may be programmed to be a different priority level. An interrupt request generation flow chart is shown in Figure 57. This flowchart is followed independently by each interrupt source.

6.4.1 INTERNAL INTERRUPT SOURCES

The internal interrupt sources are the three timers and the two DMA channels. An interrupt from each of these interrupt sources is latched in the interrupt controller. The state of the pending interrupt can be obtained by reading the interrupt request register. Also, latched DMA interrupts can be reset by the processor by writing to the interrupt request register. Note that all timers share a common bit in the interrupt request register in master mode. The interrupt controller status register may be read to determine which timer is actually causing the interrupt request. Each timer has a unique interrupt vector (see Section 6.5.1). Thus polling is not required to determine which timer has caused the interrupt in the interrupt service routine. Also, because the timers share a common interrupt control register, they are placed at a common priority level relative to other interrupt sources. Among themselves they have a fixed priority, with timer 0 as the highest priority timer and timer 2 as the lower priority timer.

6.4.2 EXTERNAL INTERRUPT SOURCES

The 80186 interrupt controller will accept external interrupt requests only when it is programmed in Master Mode. In this mode, the external pins associated
with the interrupt controller may serve either as direct interrupt inputs, or as cascaded interrupt inputs from other interrupt controllers as a programmed option. These options are selected by programming the C and SFNM bits in the INTO and INTI control registers (see Figure 50).

When programmed as direct interrupt inputs, the four interrupt inputs are each controlled by an individual interrupt control register. As stated earlier, these registers contain 3 bits which select the priority level for the interrupt and a single bit which enables the interrupt source to the processor. In addition, each of these control registers contains a bit which selects edge or level triggered mode for the interrupt input. When edge triggered mode is selected, a low-to-high transition must occur on the interrupt input before an interrupt is generated, while in level triggered mode, only a high level needs to be maintained to generate an interrupt. In edge triggered mode, the input must remain low at least 1 clock cycle before the input is rearmed. In both modes, the interrupt level must remain high until the interrupt is acknowledged, i.e., the interrupt request is not latched in the interrupt controller. The status of the interrupt input can be shown by reading the interrupt request register. Each of the external pins has a bit in this register which indicates an interrupt request on the particular pin. Note that since interrupt requests on these inputs are not latched by the interrupt controller, if the external input goes inactive, the interrupt requests (and also the bit in the interrupt request register) will also go inactive (low).

If the C (Cascade) bit of the INTO or INTI control registers is set, the interrupt input is cascaded to an external interrupt controller. In this mode, whenever the interrupt presented to the INTO or INTI line is acknowledged, the integrated interrupt controller will not provide the interrupt type for the interrupt. Instead, two INTA bus cycles will be run, with the INT2
and INT3 lines providing the interrupt acknowledge pulses for the INT0 and the INT1 interrupt requests respectively. INTO/INT2 and INT1/INT3 may be individually programmed into Cascade Mode. This allows 128 individually vectored interrupt sources if two banks of 8 external interrupt controllers each are used.

6.4.3 SLAVE MODE INTERRUPT SOURCES

When the interrupt controller is configured in Slave Mode, it accepts interrupt requests only from the integrated peripherals. Any external interrupt requests go through an external interrupt controller. This external interrupt controller requests interrupt service directly from the 80186 CPU through the INTO line. In this mode, the function of this line is not affected by the integrated interrupt controller. In addition, in Slave Mode the integrated interrupt controller must request interrupt service through this external interrupt controller. This interrupt request is made on the INT3 line (see Section 6.6.4 on external interrupt connections).

6.5 Interrupt Response

The 80186 can respond to an interrupt in two different ways. The first will occur if the internal controller is providing the interrupt vector information with the controller in Master Mode. The second will occur if the CPU reads interrupt type information from an external interrupt controller or if the interrupt controller is in Slave Mode. In both of these instances the interrupt vector information driven by the 80186 integrated interrupt controller is not available outside the 80186 microprocessor.

In each interrupt mode, when the integrated interrupt controller receives an interrupt response, the interrupt controller will automatically set the in-service bit and reset the interrupt request bit. In addition, unless the interrupt control register for the interrupt is set in Special Fully Nested Mode, the interrupt controller will prevent any interrupts from occurring from the same interrupt line until the in-service bit for that line has been cleared.

6.5.1 INTERNAL VECTORIZATION, MASTER MODE

In Master Mode, the interrupt types associated with all the interrupt sources are fixed and unalterable. These interrupt types are given in Table 5. In response to an internal CPU interrupt acknowledge the interrupt controller will generate the vector address rather than the interrupt type. On the 80186 (like the 8086) the interrupt vector address is the interrupt type multiplied by 4.

In Master Mode, no external interrupt controller need know when the integrated controller is providing an interrupt vector, nor when the interrupt acknowledge is taking place. As a result, no interrupt acknowledge bus cycles will be generated. The first external indication that an interrupt has been acknowledged will be the processor reading the interrupt vector from the interrupt vector table in memory.

Because two interrupt acknowledge cycles are not run, interrupt response to an internally vectored interrupt is 42 clock cycles. This is faster than the interrupt response when external vectoring is required, or if the interrupt controller is run in Slave Mode.

If two interrupts of the same programmed priority occur, the default priority scheme (as shown in Table 4) is used.

6.5.2 INTERNAL VECTORIZATION, SLAVE MODE

In Slave Mode, the interrupt types associated with the various interrupt sources are alterable. The upper 5 most significant bits are taken from the interrupt vector register, and the lower 3 significant bits are taken from the priority level of the device causing the interrupt. Because the interrupt type, rather than the interrupt vector address, is given by the interrupt controller in this mode the interrupt vector address must be calculated by the CPU before servicing the interrupt.

In Slave Mode, the integrated interrupt controller will present the interrupt type to the CPU in response to the two interrupt acknowledge bus cycles run by the processor. During the first interrupt acknowledge cycle, the external master interrupt controller determines which slave interrupt controller will place its interrupt vector on the microprocessor bus. During the second interrupt acknowledge cycle, the processor reads the interrupt vector from its bus. Thus, these two interrupt acknowl-
edge cycles must be run, since the integrated controller will present the interrupt type information only when the external interrupt controller signals the integrated controller that it has the highest pending interrupt request (see Figure 59). The 80186 samples the SLAVE SELECT line (INT1) during the falling edge of the clock at the beginning of T3 of the second interrupt acknowledge cycle. This input must be stable before and after this edge.

These two interrupt acknowledge cycles will be run back to back, and will be LOCKED with the LOCK output active. The two interrupt acknowledge cycles will always be separated by two idle T states, and wait states will be inserted into the interrupt acknowledge cycle if a ready is not returned by the processor bus interface. The two idle T states are inserted to allow compatibility with an external 8259A interrupt controller.

Because the interrupt acknowledge cycles must be run in Slave Mode and the integrated controller presents an interrupt type rather than a vector address, the interrupt response time is the same as for an externally vectored interrupt, namely 55 CPU clocks.

6.5.3 EXTERNAL VECTORIZATION

External interrupt vectoring occurs whenever the 80186 interrupt controller is placed in Cascade Mode, Special Fully Nested Mode, or Slave Mode (and the integrated controller is not enabled by the external master interrupt controller). In this mode, the 80186 generates two interrupt acknowledge cycles, reading the interrupt type off the lower 8 bits of the address/data bus on the second interrupt acknowledge cycle (see Figure 59). This interrupt response is exactly the same as the 8086, so that the 8259A interrupt controller can be used exactly as it would in an 8086 system. Notice that the two interrupt acknowledge cycles are LOCKED, and that two idle T-states are always inserted between the two interrupt acknowledge bus cycles, and that wait states will be inserted in the interrupt acknowledge cycle if a ready is not returned to the processor. Also

![Figure 59. 80186 Slave Mode Interrupt Acknowledge Timing](image)

NOTES:
1. SLAVE SELECT = INT1
2. INTA = INT 2
3. Driven by external interrupt controller
4. SLAVE SELECT must be driven before Phase 2 of T2 of the second INTA cycle
5. SLAVE SELECT read by 80186
notice that the 80186 provides two interrupt acknowledge signals, one for interrupts signaled by the INT0 line, and one for interrupts signaled by the INT1 line (on the INT2/INTA0 and INT3/INTA1 lines, respectively). These two interrupt acknowledge signals are mutually exclusive. Interrupt acknowledge status will be driven on the status lines (S0-S2) when either INT2/INTA0 or INT3/INTA1 signal an interrupt acknowledge.

6.5.4 EFFECT OF LOCK PREFIX ON INTERRUPT ACKNOWLEDGE CYCLES

When the interrupt controller is operating in either the cascade or slave modes and an interrupt occurs during an instruction that has been LOCKED by software, the LOCK signal timing shown in Figures 59 and 60 may be altered. Some peripheral devices used with the 80186 require contiguous INTA cycles to allow correct interrupt controller response. In such cases, the external circuitry in Figure 61 should be used to ensure that DMA or HOLD requests are blocked from stealing the bus during INTA cycles.

6.6 Interrupt Controller External Connections

The four interrupt signals can be configured into 3 major options. These are direct interrupt inputs (with the integrated controller providing the interrupt vector), cascaded (with an external interrupt controller providing the interrupt vector), or Slave Mode. In all these modes, any interrupt presented to the external lines must remain set until the interrupt is acknowledged.
6.6.1 DIRECT INPUT MODE

When the Cascade Mode bits are cleared, the interrupt input pins are configured as direct interrupt pins (see Figure 62). Whenever an interrupt is received on the input line, the integrated controller will do nothing unless the interrupt is enabled, and it is the highest priority pending interrupt. At this time, the interrupt controller will present the interrupt to the CPU and wait for an interrupt acknowledge. When the acknowledge occurs, it will present the interrupt vector address to the CPU. In this mode, the CPU will not run any external interrupt acknowledge (INTA) cycles.

![Figure 62. 80186 Non-Cascaded Interrupt Connection](image)

6.6.2 CASCADE MODE

When the Cascade Mode bit is set and the SFNM bit is cleared, the interrupt input lines are configured in Cascade Mode. In this mode, the interrupt input line is paired with an interrupt acknowledge line. The INT2/INTA0 and INT3/INTA1 lines are dual purpose; they can function as direct input lines, or they can function as interrupt acknowledge outputs. INT2/INTA0 provides the interrupt acknowledge for an INT0 input, and INT3/INTA1 provides the interrupt acknowledge for an INT1 input. Figure 63 shows this connection.

When programmed in this mode, in response to an interrupt request on the INT0 line, the 80186 will provide two interrupt acknowledge pulses. These pulses will be provided on the INT2/INTA0 line, and will also be reflected by interrupt acknowledge status being generated on the S0–S2 status lines. The interrupt type will be read on the second pulse. The 80186 externally vectored interrupt response is covered in more detail in Section 6.5.

INT0/INT2/INTA0 and INT1/INT3/INTA1 may be individually programmed into interrupt request/acknowledge pairs, or programmed as direct inputs. This means that INT0/INT2/INTA0 may be programmed as an interrupt/acknowledge pair, while INT1 and INT3/INTA1 each provide separate internally vectored interrupt inputs.

![Figure 63. 80186 Cascade and Special Fully Nested Mode Interface](image)

When an interrupt is received on a cascaded interrupt pin, the priority mask bits and the in-service bits in the particular interrupt control register will be set. This will prevent the controller from generating an 80186 CPU interrupt request from a lower priority interrupt. Also, since the in-service bit is set, any subsequent interrupt requests on the particular interrupt input line will not cause the integrated interrupt controller to generate an interrupt request to the 80186 CPU. This means that if the external interrupt controller receives a higher priority interrupt request on one of its interrupt request lines and presents it to the 80186, it will not subsequently be presented to the 80186 CPU by the integrated interrupt controller until the in-service bit for the interrupt line has been cleared.

6.6.3 SPECIAL FULLY NESTED MODE

When both the Cascade Mode bit and the SFNM bit are set, the interrupt input lines are configured in Special Fully Nested Mode. The external interface in this mode is exactly as in Cascade Mode. The only difference is in the conditions allowing an interrupt from the external interrupt controller to the integrated interrupt controller to interrupt the 80186 CPU.

When an interrupt is received from a Special Fully Nested Mode interrupt line, it will interrupt the 80186 CPU if it is the highest priority interrupt pending regardless of the state of the in-service bit for the interrupt source in the interrupt controller. When an interrupt is acknowledged from a special fully nested mode interrupt line, the priority mask bits and the in-service bits in the particular interrupt control register will be
set into the interrupt controller's in-service and priority mask registers. This will prevent the interrupt controller from generating an 80186 CPU interrupt request from a lower priority interrupt. Unlike Cascade Mode, however, the interrupt controller will not prevent additional interrupt requests generated by the same external interrupt controller from interrupting the 80186 CPU. This means that if the external (cascaded) interrupt controller receives a higher priority interrupt request on one of its interrupt request lines and presents it to the integrated controller's interrupt request line, it may cause an interrupt to be generated to the 80186 CPU, regardless of the state of the in-service bit for the interrupt line.

If the SFNM bit is set and the Cascade Mode bit is not set, the controller will provide internal interrupt vectoring. It will also ignore the state of the in-service bit in determining whether to present an interrupt request to the CPU. In other words, it will use the SFNM conditions of interrupt generation with an internally vectored interrupt response, i.e., if the interrupt pending is the highest priority type pending, it will cause a CPU interrupt regardless of the state of the in-service bit for the interrupt. This operation is only applicable to INT0 and INT1, which have SFNM bits in their control registers.

6.6.4 SLAVE MODE

When the SLAVE/MASTER bit in the peripheral relocation register is set, the interrupt controller is in Slave Mode. In this mode, all four interrupt controller input lines are used to perform the necessary handshaking with the external master interrupt controller. Figure 64 shows the hardware configuration of the 80186 interrupt lines with an external controller in Slave Mode.

Because the integrated interrupt controller is a slave controller, it must be able to generate an interrupt input for an external interrupt controller. It also must be signaled when it has the highest priority pending interrupt to know when to place its interrupt vector on the bus. These two signals are provided by the INT3/Slave Interrupt Output and INT1/Slave Select lines, respectively. The external master interrupt controller must be able to interrupt the 80186 CPU, and needs to know when the interrupt request is acknowledged. The INT0 and INT2/INTA0 lines provide these two functions.

6.7 Example 8259A or 82C59A Cascade Mode Interface

Figure 65 shows the 80186 and 8259A (or 82C59A) in Cascade Mode. The code to initialize the 80186 interrupt controller is given in Appendix E. Notice that an interrupt ready signal must be returned to the 80186 to prevent the generation of wait states in response to the interrupt acknowledge cycles. In this configuration the INT0 and INT2 lines are used as direct interrupt input lines. Thus, this configuration provides 10 external interrupt lines: 2 provided by the 80186 interrupt controller itself, and 8 from the external 8259A. Also, the 8259A is configured as a master interrupt controller. It will only receive interrupt acknowledge pulses in response to an interrupt it has generated. It may be cascaded again to up to 8 additional 8259As (each of which would be configured in Slave Mode).

6.8 Interrupt Latency

Interrupt latency time is the time from when the 80186 receives the interrupt to the time it begins to respond to the interrupt. This is different from interrupt response time, which is the time from when the processor actually begins processing the interrupt to when it actually executes the first instruction of the interrupt service routine. The factors affecting interrupt latency are the instruction being executed and the state of the interrupt enable flip-flop. The interrupt enable flip-flop must be explicitly set by issuing the STI instruction. Since interrupt vectoring automatically clears the flip-flop, it is necessary to set the flip-flop within the interrupt service routine if nested interrupts are desired.
When interrupts are enabled in the CPU, the interrupt latency is a function of the instructions being executed. Only repeated instructions will be interrupted before being completed, and those only between their respective iterations. This means that the interrupt latency time could be as long as 69 CPU clocks, which is the time it takes the processor to execute an integer divide instruction (with a segment override prefix, see below), the longest single instruction on the 80186.

Other factors can affect interrupt latency. An interrupt will not be accepted between the execution of a prefix (such as segment override prefixes and lock prefixes) and the instruction. In addition, an interrupt will not be accepted between an instruction which modifies any of the segment registers and the instruction immediately following the instruction. This is required to allow the stack to be changed. If the interrupt were accepted, the return address from the interrupt would be placed on a stack which was not valid (the Stack Segment register would have been modified but the Stack Pointer register would not have been). An interrupt will not be accepted between the execution of the WAIT instruction and the instruction immediately following it if the TEST input is active. If the WAIT sees the TEST input inactive, however, the interrupt will be accepted, and the WAIT will be re-executed after the interrupt return. Finally, the 80C186 and 80C188 will not accept interrupts during refresh bus cycles.

7.0 CLOCK GENERATOR

The 80186 clock generator provides the main clock signal for all 80186 integrated components, and all CPU synchronous devices in the 80186 system. This clock generator includes a crystal oscillator, divide by two counter, reset circuitry, and ready generation logic. A block diagram of the clock generator is shown in Figure 66.

7.1 Crystal Oscillator

All 80186 family microprocessors use a parallel resonant Pierce oscillator. For all NMOS 80186/80188 applications and lower frequency 80C186/80C188 applications, a fundamental mode crystal is appropriate. At higher frequencies, the diminishing thickness of fundamental mode crystals makes a third overtone crystal the appropriate choice. The addition of external capacitors at X1 and X2 is always required, and a third overtone crystal also requires an RC tank circuit to select the third overtone frequency over the fundamental frequency (see Figure 67).

The recommendations given in the 80186 family product data sheets for the values of the external components should be taken only as guidelines since there are situations where the oscillator operation can be modified somewhat. One example would be the case where the circuit layout introduces significant stray capacitance to the X1 and X2 pins. Another example is at low frequencies (CLKOUT less than 6 MHz) where slightly
larger capacitors are desirable. Finally, it is also possible to use ceramic resonators in place of crystals for low cost when precise frequencies are not required.

For assistance in selecting the external oscillator components for unusual circumstances, the best resource is the crystal manufacturer. The foremost circuit consideration is that the oscillator start correctly over the entire voltage and temperature ranges expected in operation.

Figure 66. 80186 Clock Generator Block Diagram

Figure 67. 80186 Family Crystal Connections

Figure 68. 80186 Clock Generator Reset

NOTES:
1. RES sampled on rising edge of oscillator input signal (X1).
2. Internal clear pulse generated.
3. Internal clear pulse drives CLKOUT high, resynchronizing the clock generator.
4. RESET output goes active (TCLR0).
5. RES allowed to go inactive after minimum 4 CLKOUT cycles, recognized at rising CLKOUT.
6. RESET output goes inactive 1½ CLKOUT cycles plus TCLR0 after recognition of RES inactive.
7. First instruction prefetch occurs 6½ CLKOUT cycles after coming out of reset.
7.2 Using an External Oscillator

An external oscillator may be used with the 80186. The external frequency input (EFI) signal is connected directly to the X1 input of the oscillator. X2 should be left not connected. This oscillator input drives an internal divide-by-two counter to generate the CPU clock signal, so the external frequency input can be of practically any duty cycle, so long as the minimum high and low times for the signal (as stated in the data sheet) are met.

7.3 Clock Generator

The output of the crystal oscillator (or the external frequency input) generates a 50% duty cycle clock for the 80186 system. All 80186 timing is referenced to this signal, which is available on the CLKOUT pin of the 80186. This signal will change state on the high-to-low transition of the EFI signal.

7.4 Ready Generation

The clock generator also includes the circuitry required for ready generation. Interfacing to the SRDY and ARDY inputs this provides is covered in Section 3.1.6.

7.5 Reset

The 80186 clock generator also provides a synchronized reset signal for the system. This signal is generated from the reset input (RES) to the 80186. The clock generator synchronizes this signal to the clockout signal.

The reset input also resets the divide-by-two counter. A one clock cycle internal clear pulse is generated when the RES input signal goes active. This clear pulse goes active beginning on the first low-to-high transition of the X1 input after RES goes active, and goes inactive on the next low-to-high transition of the X1 input. In order to insure that the clear pulse is generated on the next oscillator cycle, the RES input signal must satisfy a setup time to the high-to-low oscillator input signal (see Figure 68). During this clear, CLKOUT will be high. On the next high-to-low transition of X1, CLKOUT will go low, and will change state on every subsequent high-to-low X1 transition.

The reset signal presented to the rest of the 80186, and also the signal present on the RESET output pin of the 80186 is synchronized by the high-to-low transition of the clockout signal of the 80186. This signal remains active as long as the RES input also remains active. After the RES input goes inactive, the 80186 will begin to fetch its first instruction (at memory location FFFF0H) after 6 1/2 CPU clock cycles (i.e., T1 of the first instruction fetch will occur 6 1/2 clock cycles later). To ensure that the RESET output will go inactive on the next CPU clock cycle, the inactive going edge of the RES input must satisfy certain hold and setup times to the low-to-high edge of the CLKOUT signal of the 80186 (see Figure 68).

8.0 CHIP SELECTS

The 80186 includes a chip select unit which provides hardware chip select signals for memory and I/O accesses generated by the 80186 CPU and DMA units. This unit is programmable such that it can fulfill the chip select requirements (in terms of memory device or bank size and speed) of most small and medium sized 80186 systems.

The chip selects are driven only for internally generated bus cycles. Any cycles generated by an external unit (e.g., an external DMA controller) will not cause the chip selects to go active. Thus, any external bus masters must be responsible for their own chip select generation. Also, during a bus HOLD, the 80186 does not float the chip select lines. Therefore, logic must be included to enable the devices which the external bus master wishes to access (see Figure 69).
8.1 Memory Chip Selects

The 80186 provides six discrete memory chip select lines. These signals are named UCS, LCS, and MCS0-3 for Upper Memory Chip Select, Lower Memory Chip Select and Midrange Memory Chip Select 0–3. They are meant (but not limited) to be connected to the three major areas of the 80186 system memory (see Figure 70).

The upper limit of UCS and the lower limit of LCS are fixed at FFFFFH and 00000H in memory space, respectively. The other limit is set by the memory size programmed into the control register for the chip select line. Mid-range memory allows both the base address and the block size of the memory area to be programmed. The only limitation is that the base address must be programmed to be an integer multiple of the total block size. For example, if the block size was 128K bytes (4 32K byte chunks) the base address could be 0 or 20000H, but not 10000H.

The memory chip selects are controlled by 4 registers in the peripheral control block (see Figure 71). These include 1 each for UCS and LCS, the values of which determine the size of the memory blocks addressed by these two lines. The other two registers are used to control the size and base address of the mid-range memory block.

On reset, only UCS is active. It is programmed to be active for the top 1K memory block, to insert 3 wait states to all memory fetches, and to factor external ready for every memory fetch (see Section 8.3 for more information on internal ready generation). None of the other chip select lines will be active until all necessary registers for a signal have been accessed (not necessarily written, a read to an uninitialized register will enable the chip select function controlled by that register).

8.2 Peripheral Chip Selects

The 80186 provides seven discrete chip select lines which are meant to be connected to peripheral components in an 80186 system. Each of these lines is active for one of seven continuous 128 byte areas in memory or I/O space above a programmed base address.
The peripheral chip selects are controlled by two registers in the internal peripheral control block (see Figure 71). These registers set the base address of the peripherals and map the peripherals into memory or I/O space. Both of these registers must be accessed before any of the peripheral chip selects will become active.

A bit in the MPCS register allows PCS5 and PCS6 to become latched A1 and A2 outputs. When this option is selected, PCS5 and PCS6 reflect the state of A1 and A2 throughout a bus cycle. These allow external peripheral register selection in a system in which the addresses are not latched. Upon reset, these lines are driven high.

8.3 Ready Generation

The 80186 includes a ready generation unit. This unit generates an internal ready signal for all accesses to memory or I/O areas to which the chip select circuitry of the 80186 responds.

For each ready generation area, 0–3 wait states may be inserted by the internal unit. Table 5 shows how the ready control bits should be programmed to provide this. In addition, the ready generation circuit may be programmed to ignore or include the state of the external ready pins. When using both internal and external ready generation, both elements must be fulfilled before a busy cycle will end. The external ready condition is always required upon reset for accesses involving the top 1K of memory. Therefore, at least one of the ready pins must be connected to functional ready circuitry or be tied HIGH until UCS is reprogrammed early in the initialization sequence.

### Table 5. 80186 Wait State Programming

<table>
<thead>
<tr>
<th>R2</th>
<th>R1</th>
<th>R0</th>
<th>Number of Wait States</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 + external ready</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1 + external ready</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2 + external ready</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3 + external ready</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 (no external ready required)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1 (no external ready required)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2 (no external ready required)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3 (no external ready required)</td>
</tr>
</tbody>
</table>

8.4 Examples of Chip Select Usage

Many examples using the chip select lines are given in the bus interface section of this note (Section 3.2). The key point to remember when using the chip select function is that they are only activated during bus cycles generated by the 80186. When another master has the bus, it must generate its own chip selects. In addition, whenever the bus is given by the 80186 to an external master (through HOLD/HLDA) the 80186 does not float the chip select lines.

8.5 Overlapping Chip Select Areas

Generally, the chip selects of the 80186 should not be programmed such that any two areas overlap. In addition, none of the programmed chip select areas should overlap any locations of the integrated 256-byte control register block. The consequences of doing this are:

Whenever two chip select lines are programmed to respond to the same area, both will be activated during any access to that area. When this is done, the ready bits for both areas must be programmed to the same value. If this is not done, the processor response to an access in this area is indeterminate. This rule also applies to overlapping chip selects with the integrated control block.

If any of the chip select areas overlap the integrated 256-byte control block, the timing on the chip select line is altered. An access to the control block will temporarily activate the corresponding chip select pin, but it will go inactive prematurely.

8.6 MCS Functionality and the 80C186

The 80C186 MCS0, MCS1 and MCS3 pins change function when the part is configured for Enhanced Mode (see Section 9.0 for an explanation of Enhanced Mode). The 80C188 MCS pins function the same in both modes. These pins are configured to support an asynchronous numerics floating point coprocessor (see Table 6). Thus, the 80C186 does not provide the complete range of middle chip selects normally available. However, the functionality of the MCS2 pin and the programming features of the MPCS and MMCS registers are still available.

### Table 6. MCS Pin Definitions

<table>
<thead>
<tr>
<th>Pin #</th>
<th>Compatible Mode</th>
<th>Enhanced Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>MCS3</td>
<td>NPS, Numerics Processor Select</td>
</tr>
<tr>
<td>36</td>
<td>MCS2</td>
<td>MCS2</td>
</tr>
<tr>
<td>37</td>
<td>MCS1</td>
<td>ERROR, Numerics Processor Error</td>
</tr>
<tr>
<td>38</td>
<td>MCS0</td>
<td>PEREQ, Processor Extension Request</td>
</tr>
</tbody>
</table>

In Enhanced Mode, it is still possible to program the starting address, block size and ready requirements of the middle chip selects. This allows the user to take advantage of the wait-state generation logic on the 80C186 even though the majority of external chip selects are not active. It is also possible to use MCS2 which is active for one fourth the block size (see Figure 72).
9.0 80C186 PRODUCT ENHANCEMENTS

The 80C186 and 80C188 are for the most part identical to their NMOS counterparts, and may be used interchangeably. However, aside from the fact that the 80C186 and 80C188 are designed with Intel's CHMOS III technology and provide greater operating frequencies and less power consumption, they also provide two new operating units not found on the 80186 or 80188: the Refresh Control Unit and the Power-Save Unit. To ensure that the new features of the 80C186 are not accidentally programmed in older designs, the 80C186 has two operating modes: Compatible Mode and Enhanced Mode. Compatible Mode implies that the register, programming and pin definition of the 80C186 is identical to that of the 80186. Enhanced Mode implies that the 80C186 provides a super-set of functionality to that of the 80186.

The different modes are selected during RESET. The timing diagram in Figure 73 shows how the 80C186 samples the TEST input pin just before and just after RES is removed to determine if the device will enter Enhanced Mode. Tying the RESET output pin back to the TEST input pin ensures that the 80C186 or 80C188 enters enhanced mode. If the TEST input is used for external synchronization of code, then RESET can be OR'ed with the other input provided it is always active (low) just after RESET.

When the 80C186 (not the 80C188) is in Enhanced Mode, some of the MCS chip select lines change functionality to support an asynchronous numerics floating-point coprocessor. Refer to Section 8.6 for more detail.

9.1 Refresh Control Unit

To simplify the design of a dynamic memory controller, the 80C186 incorporates integrated address and clock counters which, along with the BIU, facilitate dynamic memory refreshing. A block diagram of the Refresh Control Unit (RCU) and its relationship to the BIU is shown in Figure 74. To the memory interface, a refresh request looks exactly like a memory read bus cycle. This is because a refresh bus cycle is a memory read operation. Because the RCU is integrated into the 80C186, functions such as chip selects and wait-state control can be used effectively.
The 9-bit counter is controlled by the BIU and is used whenever a refresh bus cycle is executed. Thus, any dynamic memory whose refresh address requirement does not exceed nine bits can be directly supported by the 80C186. The 9-bit address counter along with a 6-bit base register define a full 20-bit refresh address. The 9-bit counter generates a signal to initiate a refresh bus cycle. When the counter decrements to 1 (it is decremented every clock cycle), a refresh request is presented to the BIU. When the bus is free, the BIU will run the refresh (memory read) bus cycle. Refresh requests have a higher priority than any other bus request (i.e., CPU, DMA, HOLD).

9.1.1 REFRESH CONTROL UNIT PROGRAMMING

There are several registers in the Peripheral Control Block that control the RCU. These registers are only accessible when the 80C186 or 80C188 are operating in Enhanced mode. Otherwise, a read or write to these registers is ignored.

The three control registers are MDRAM, CDRAM, and EDRAM (see Figure 75). These registers define the operating characteristics of the RCU. The MDRAM register programs the base address (upper 7 bits) of the refresh address (see Figure 76). This allows the refresh address to be mapped into any 4 kilobyte boundary within the 1 megabyte 80C186 address space. The MDRAM register is not altered whenever the refresh address bits (A1 through A9 in Figure 76) roll over. In other words, the refresh address does not act like a linear counter found in a typical DMA controller.

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>0</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>E4H</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E2H</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E0H</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Notes:**
  1. Bits 0–8: T0–T8, Refresh request down counter clock count. These bits are read only and represent the current value of the counter. Any write operation to these bits is ignored.
  2. Bit 15: E, enables the operation of the refresh control unit.
  3. Bits 9–15: MO–MS, are used to define address bits A13–A19 (respectively) of the 20-bit memory address. These bits are set to zero on RESET.
Address Bit | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0

Bit 0: Always driven to 1 (Logic High). This is true for both the 80C186 and the 80C188.
Bits 1-9: CA0-CAB, are generated by the 9-bit Linear-Feedback shift counter.
Bits 10-12: Always driven to 0 (Logic Low).
Bits 13-19: M0-M6, are defined by the MDRAM Register.

Figure 76. Physical Address Generation

The CDRAM register defines the time interval between refresh requests by initializing the value loaded into the 9-bit down counter. Thus, the higher the value, the longer the amount of time between requests. The down-counter is decremented every falling edge of CLKOUT, regardless of the activity of the CPU or BIU. When the counter decrements to 1, a request is generated and the counter is again loaded with the value in the CDRAM register. The amount of time between refresh requests can be calculated using the equation shown in Figure 77. The minimum value that can be programmed into the CDRAM register is 18 (12H) regardless of the operating frequency. This is due to the minimum number of clocks required between each successive request to ensure the BIU has enough time to execute the refresh bus cycle. The BIU is not capable of queueing requests; if another request is generated before the current request is executed, the current request is lost. This applies only to the request itself, not the address associated to the request. The refresh address is only changed after the BIU has run the bus cycle. Thus it is possible to miss refresh requests, but not refresh addresses.

The EDRAM register has two functions, depending on whether it is being written or read. During writes to the EDRAM register, only the Enable bit is active. Setting the Enable bit enables the RCU while clearing the Enable bit disables the RCU. Whenever the RCU is enabled, the contents of the CDRAM register are loaded into the 9-bit down counter and refresh requests will be generated when the counter reaches 1. Disabling the RCU stops and clears the counter. A read of the EDRAM register will return the current value of the Enable bit as well as the current value of the 9-bit down counter (zero if the RCU is not enabled). Writing to EDRAM register when the RCU is running does not modify the count value in the 9-bit counter.

9.1.2 REFRESH CONTROL UNIT OPERATION

Figure 78 illustrates the two major functions of the refresh control unit that are responsible for initiating and controlling the refresh bus cycles.

The down counter is loaded on the falling edge of CLKOUT, when either the Enable bit is set or the counter decrements to 1. Once loaded, the down counter will decrement every falling edge of CLKOUT (as long as the Enable bit remains set).

When the counter decrements to 1, two things happen. First, a request is generated to the BIU to run a refresh bus cycle. The request remains active until the bus cycle is run. Second, the down counter is reloaded with the value contained in the CDRAM register. At this time, the down counter will again begin counting down every clock cycle. It does not wait until the request has been serviced. This is done to ensure that each refresh request occurs at the correct interval. Otherwise, the time between refresh requests would also be a function of bus activity, which is unpredictable. When the BIU services the refresh request, it will clear the request and increment the refresh address.

R_{PERIOD} (\mu s) \times \text{FREQ (MHz)} = \text{CDRAM Register Value}

\[
R_{\text{PERIOD}} = \frac{\text{Maximum Refresh period specified by the DRAM manufacturer (time in microseconds).}}{\text{FREQ} = \text{Operating Frequency at 80C186 in MHz.}}
\]

\[
\text{# Refresh Rows} = \text{Total number of rows to be refreshed.}
\]

\[
\% \text{ Overhead} = \text{Derating factor to compensate for missed refresh requests (typically 1–5%).}
\]

Figure 77. Equation to Calculate Refresh Interval
9.1.3 REFRESH ADDRESS CONSIDERATIONS

The physical address that is generated during a refresh bus cycle is shown in Figure 76, and applies to both the 80C186 and 80C188. The refresh address bits CA0 through CA8 are generated using a linear-feedback shift counter which does not increment the addresses linearly from 0 through FFFH (although they do follow a predictable algorithm). Further, note that for the 80C188, address bit A0 does not toggle during refresh operation, which means that it cannot be used as part of the refresh address applied to the dynamic memory device. Typically, A0 is used as part of memory decoding in 80C188 applications, unlike the 80C186 which uses A0 along with BHE to select an upper or lower bank. Therefore, when designing with the 80C188, it is important not to include A0 as part of the row address that is used for refreshing. Appendix K illustrates memory address multiplexing techniques that can be applied to the 80C186 and 80C188.

9.1.4 REFRESH OPERATION AND BUS HOLD

When another bus master has control of the bus, the HLDA signal is kept active as long as the HOLD input remains active. If a refresh request is generated while HOLD is active, the 80C186 will remove (drive inactive) the HLDA signal to indicate to the current bus master that the 80C186 wishes to regain control of the bus (see Figure 79). Only when the HOLD input is removed will the BIU begin the refresh bus cycle.

Therefore, it is the responsibility of the system designer to ensure that the 80C186 can regain the bus if a refresh request is signalled. The sequence of HLDA going inactive while HOLD is active can be used to signal a pending refresh request. HOLD need only go inactive for one clock period to allow the refresh bus cycle to be run. If HOLD is again asserted, the 80C186 will give up the bus after the refresh bus cycle has been run (provided there is not another refresh request generated during that time).

9.2 Power-Save Unit

The Power-Save Unit is intended to benefit applications by lower power consumption while maintaining regular operation of the CPU. The 80C186 Power-Save mechanism lowers current needs by reducing the operating frequency.

The Power-Save Unit is an internal clock divider as shown in Figure 80. Because the Power-Save Unit will change the internal operating frequency, all other units within the 80C186 will be affected by the clock change. This includes the CPU, Timers, Refresh, DMA, and BIU. Thus, by using the Power-Save feature, the net effect is similar to changing the input clock frequency.
NOTES:
1. System generates HOLD request.
2. HLDA is returned and 80C186 floats the bus/control lines.
3. Refresh request is generated by the 80C186.
4. 80C186 lowers (removes) HLDA to signal that it wants the bus back.
5. 80C186 waits until HOLD is lowered (removed) for at least 1 clock cycle (minimum HOLD setup and hold time) to execute the refresh bus cycle. If HOLD is never lowered, the 80C186 will not take over the bus.
6. 80C186 runs the refresh bus cycle.
7. HOLD can be again asserted after the 1 clock duration.
8. The refresh request is cleared after the bus cycle has been executed.
9. If HOLD was again asserted, the 80C186 will immediately relinquish the bus back. If no HOLD occurred, normal CPU operation will resume.

Figure 79. HOLD/HLDA Timing and Refresh Request

9.2.1 POWER-SAVE UNIT PROGRAMMING

The PD CON register (see Figure 81) controls the operation of the Power-Save Unit. This register is available for programming when the 80C186 or 80C188 is in Enhanced Mode. Reads or write to the PD CON register in Compatible Mode result in no operation, and the value returned will be all ones.

When the Enable bit in the PD CON register is set, the Power-Save Unit is active and, depending on the condition of the F0 and F1 bits, the operating clock of the 80C186 is changed from normal operation. When the Enable bit is cleared, the 80C186 will operate at the standard divide by 2 clock rate. The Enable bit is automatically cleared whenever a non-masked interrupt occurs. Thus, if the Power-Save feature is enabled and an unmasked interrupt of sufficient priority is received, the Enable bit clears and the processor executes at full speed. This allows interrupts to be processed at full speed. A return from the interrupt does not automatically set the Enable bit. This must be done as part of the interrupt routine. Software interrupts do not clear the Enable bit.

Figure 80. Simplified Power-Save Internal Operation
The F0 and F1 bits determine the divisor of the Power-Save unit. Figure 81 provides a list of the various combinations of the bits and their division factor. Note that the divisor is related to the output clock, not the input clock at pin X1. Selecting a divisor of 1 does not reduce the power consumption. The operating clock of the 80C186 must not be divided below the minimum operating frequency specified in data sheet (500 kHz). Figure 81 also indicates the minimum operating frequency required in order to use a specific divisor.

9.2.2 POWER-SAVE OPERATION

When the Enable bit in the PDCON register is set, the clock divider circuitry will turn on during the write to the PDCON register (refer to Figure 82). At the falling edge of T3 of the register write, CLKOUT will change to reflect the new divisor. If any values of F0–F1 other than zero have been programmed, the CLKOUT period will be increased over undivided CLKOUT, starting with the low phase. CLKOUT will not glitch.

The Power-Save Unit remains active until one of three events happens: either the Enable bit in the PDCON register is cleared, new values for F0 and F1 are programmed, or an unmasked interrupt is received. In the first two cases, the changes directly follow Figure 82.

When an unmasked interrupt is received, the operating frequency is changed as shown in Figure 82, but may occur at any T3 bus state in progress at the time of the interrupt. Thus, it is not possible to determine exactly when, in the event of an interrupt, the Power-Save unit will be disabled.

10.0 SOFTWARE IN AN 80186 SYSTEM

Since the 80186 is object code compatible with the 8086 and 8088, the software in an 80186 system is very similar to that in an 8086 system. Because of the hardware chip select functions, however, a certain amount of initialization code must be included when using those functions on the 80186.

10.1 System Initialization in an 80186 System

The 80186 includes circuitry which directly affects the ability of the system to address memory and I/O devices, namely the chip select circuitry. This circuitry must be initialized before the memory areas and peripheral devices addressed by the chip select signals can be used.
Upon reset, the UMCs register is programmed to be active for all memory fetches within the top 1K byte of memory space. It is also programmed to insert three wait states to all memory accesses within this space. If the hardware chip selects are used, they must be programmed before the processor leaves this 1K byte area of memory. If a jump to an area for which the chips are not selected occurs the processor will fetch garbage. Appendix F shows a typical initialization sequence for the 80186 chip select unit.

10.2 Instruction Execution Differences between the 8086 and 80186

There are a few instruction execution differences between the 8086 and the 80186. These differences are:

**UNDEFINED OPCODES:**

When the opcodes 63H, 64H, 65H, 66H, 67H, F1H, FEH XX111XXXB and FFH XX111XXXB are executed, the 80186 will execute an illegal instruction exception, interrupt type 6. The 8086 will ignore the opcode.

**0FH OPCODE:**

When the opcode 0FH is encountered, the 8086 will execute a POP CS, while the 80186 will execute an illegal instruction exception, interrupt type 6.

**WORD WRITE AT OFFSET FFFFH:**

When a word write is performed at offset FFFFH in a segment, the 8086 will write one byte at offset FFFFH, and the other at offset 0, while the 80186 will write one byte at offset FFFFH, and the other at offset 10000H (one byte beyond the end of the segment). One byte segment underflow will also occur (on the 80186) if a stack PUSH is executed and the Stack Pointer contains the value 1.

**SHIFT/ROTATE BY VALUE GREATER THAN 31:**

Before the 80186 performs a shift or rotate by a value (either in the CL register, or by an immediate value) it ANDs the value with 1FH, limiting the number of bits rotated to less than 32. The 8086 does not do this.

**LOCK PREFIX:**

The 8086 activates its LOCK signal immediately after executing the LOCK prefix. The 80186 does not activate the LOCK signal until the processor is ready to begin the data cycles associated with the LOCKed instruction.

**NOTE:**

When executing more than one LOCKed instruction, always make sure there are 6 bytes of code between the end of the first LOCKed instruction and the start of the second LOCKed instruction.

**INTERRUPTED STRING MOVE INSTRUCTIONS:**

If an 8086 is interrupted during the execution of a repeated string move instruction, the return value it will push on the stack will point to the last prefix instruction before the string move instruction. If the instruction had more than one prefix (e.g., a segment override prefix in addition to the repeat prefix), it will not be re-executed upon returning from the interrupt. The 80186 will push the value of the first prefix to the repeated instruction, so long as prefixes are not repeated, allowing the string instruction to properly resume.

**CONDITIONS CAUSING DIVIDE ERROR WITH AN INTEGER DIVIDE:**

The 8086 will cause a divide error whenever the absolute value of the quotient is greater than 7FFFH (for word operations) or if the absolute value of the quotient is greater than 7FH (for byte operations). The 80186 has expanded the range of negative numbers allowed as a quotient by 1 to include 8000H and 80H. These numbers represent the most negative numbers representable using 2's complement arithmetic (equaling −32768 and −128 in decimal, respectively).

**ESC OPCODE:**

The 80186 may be programmed to cause an interrupt type 7 whenever an ESCape instruction (used for co-processors like the 8087) is executed. The 8086 has no such provision. Before the 80186 performs this trap, it must be programmed to do so.

These differences can be used to determine whether the program is being executed on an 8086 or an 80186. Probably the safest execution difference to use for this purpose is the difference in multiple bit shifts. For example, if a multiple bit shift is programmed where the shift count (stored in the CL register) is 33, the 8086 will shift the value 33 bits, whereas the 80186 will shift it only a single bit.

In addition to the instruction execution differences noted above, the 80186 includes a number of new instruction types, which simplify assembly language programming of the processor, and enhance the performance of higher level languages running on the processor. These new instructions are covered in depth in the 8086/80186 users manual and in Appendix H of this note.
APPENDIX A
PERIPHERAL CONTROL BLOCK

All the integrated peripherals within the 80186 microprocessor are controlled by sets of registers contained within an integrated peripheral control block. The registers are physically located within the peripheral devices they control, but are addressed as a single block of registers. This set of registers encompasses 256 contiguous bytes and can be located on any 256 byte boundary of the 80186 memory or I/O space. Maps of these registers are shown in Figure A-1 for the 80186/80188 and in Figure A-2 for the 80C186/80C188. Any unused bytes are reserved.

A.1 SETTING THE BASE LOCATION OF THE PERIPHERAL CONTROL BLOCK

In addition to the control registers for each of the integrated 80186 peripheral devices, the peripheral control block contains the peripheral control block relocation register. This register allows the peripheral control block to be re-located on any 256 byte boundary within the processor's memory or I/O space. Figure A-2 shows the layout of this register.

This register is located at offset FEH within the peripheral control block. Since it is itself contained within the peripheral control block, any time the location of the peripheral control block is moved, the location of the relocation registers will also move.

In addition to the peripheral control block relocation information, the relocation register contains two additional bits. One is used to set the interrupt controller into slave mode. The other is used to force the processor to trap whenever an ESCape (coprocessor) instruction is encountered.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEH</td>
<td>Relocation Register</td>
</tr>
<tr>
<td>DAH</td>
<td>DMA Descriptors Channel 1</td>
</tr>
<tr>
<td>D0H</td>
<td>DMA Descriptors Channel 0</td>
</tr>
<tr>
<td>CAH</td>
<td>Chip-Select Control Registers</td>
</tr>
<tr>
<td>C0H</td>
<td></td>
</tr>
<tr>
<td>ABH</td>
<td>Timer 2 Control Registers</td>
</tr>
<tr>
<td>A0H</td>
<td>Timer 1 Control Registers</td>
</tr>
<tr>
<td>66H</td>
<td>Timer 0 Control Registers</td>
</tr>
<tr>
<td>60H</td>
<td></td>
</tr>
<tr>
<td>5EH</td>
<td></td>
</tr>
<tr>
<td>58H</td>
<td></td>
</tr>
<tr>
<td>56H</td>
<td></td>
</tr>
<tr>
<td>50H</td>
<td></td>
</tr>
<tr>
<td>3EH</td>
<td>Interrupt Controller Registers</td>
</tr>
<tr>
<td>20H</td>
<td></td>
</tr>
</tbody>
</table>

Figure A-1. 80186/80188 Integrated Peripheral Control Block
Because the relocation register is contained within the peripheral control block, upon reset the relocation register is automatically programmed with the value 20FFH. This means that the peripheral control block will be located at the very top (FF00H to FFFFH) of I/O space. Thus, after reset the relocation register will be located at word location FFFEH in I/O space.

To relocate the peripheral control block to the memory range 10000H-100FFH, for example, the user programs the relocation register with the value 1100H. Since the relocation register is contained within the peripheral control block, it moves to word location 100FEH in memory space.
Whenever mapping the 188 peripheral control block to another location, the programming of the relocation register should be done with a byte write (i.e., OUT DX, AL). Any access to the control block is done 16 bits at a time. Thus, internally, the relocation register will get written with 16 bits of the AX register while externally, the BIU will run only one 8 bit bus cycle. If a word instruction is used (i.e., OUT DX, AX), the relocation register will be written on the first bus cycle. The BIU will then run a second bus cycle which is unnecessary. The address of the second bus cycle will no longer be within the control block (i.e., the control block was moved on the first cycle), and therefore, will require the generation of an external ready signal to complete the cycle. For this reason we recommend byte operations to the relocation register. Byte instructions may also be used for the other registers in the control block and will eliminate half of the bus cycles required if a word operation had been specified. Byte operations are only valid on even addresses though, and are undefined on odd addresses.

A.2 Peripheral Control Block Registers

Each of the integrated peripherals' control and status registers are located at a fixed location above the programmed base location of the peripheral control block. There are many locations within the peripheral control block which are not assigned to any peripheral. If a write is made to any of these locations, the bus cycle will be run, but the value will not be stored in any internal location. This means that if a subsequent read is made to the same location, the value written will not be read back.

The processor will run an external bus cycle for any memory or I/O cycle which accesses a location within the integrated control block. This means that the address, data, and control information will be driven on the 80186 external pins just as if a "normal" bus cycle had been run. Any information returned by an external device will be ignored, however, even if the access was to a location which does not correspond to any of the integrated peripheral control registers. The above is also true for the 80188, except that the word access made to the integrated registers will be performed in a single bus cycle internally, while externally, the BIU runs two bus cycles.

The processor internally generates a ready signal whenever any of the integrated peripherals are accessed; thus any external ready signals are ignored. This ready will also be returned if an access is made to a location within the 256 byte area of the peripheral control block which does not correspond to any integrated peripheral control register. The processor will insert 0 wait states to any access within the integrated peripheral control block except for accesses to the timer registers. Any access to the timer control and counting registers will incur 1 wait state. This wait state is required to properly multiplex processor and counter element accesses to the timer control registers.

All accesses made to the integrated peripheral control block will be word accesses. Any write to the integrated registers will modify all 16 bits of the register, whether the opcode specified a byte write or a word write. A byte read from an even location should cause no problems, but the data returned when a byte read is performed from an odd address within the peripheral control block is undefined. This is true both for the 80186 and the 80188. As stated above, even though the 80188 has an external 8 bit data bus, internally it is still a 16-bit machine. Thus, the word accesses performed to the integrated registers by the 80188 will each occur in a single bus cycle internally while externally the BIU runs two bus cycles. The DMA controller cannot be used for either read or write accesses to the peripheral control block.
80186 SYNCHRONIZATION INFORMATION

Many input signals to the 80186 are asynchronous, that is, a specified set up or hold time is not required to insure proper functioning of the device. Associated with each of these inputs is a synchronizer which samples this external asynchronous signal, and synchronizes it to the internal 80186 clock.

B.1 WHY SYNCHRONIZERS ARE REQUIRED

Every data latch requires a certain set up and hold time in order to operate properly. At a certain window within the specified set up and hold time, the part will actually try to latch the data. If the input makes a transition within this window, the output will not attain a stable state within the given output delay time. The size of this sampling window is typically much smaller than the actual window specified by the data sheet, however part to part variation could move this window around within the specified window in the data sheet.

Even if the input to a data latch makes a transition while a data latch is attempting to latch this input, the output of the latch will attain a stable state after a certain amount of time, typically much longer than the normal strobe to output delay time. Figure B-1 shows a normal input to output strobed transition and one in which the input signal makes a transition during the latch’s sample window. In order to synchronize an asynchronous signal, all one needs to do is to sample the signal into one data latch long enough for the output to stabilize, then latch it into a second data latch. Since the time between the strobe into the first data latch and the strobe into the second data latch allows the first data latch to attain a steady state (or to resolve the asynchronous signal), the second data latch will be presented with an input signal which satisfies any set up and hold time requirements it may have.

Thus, the output of this second latch is a synchronous signal with respect to its strobe input.

A synchronization failure can occur if the synchronizer fails to resolve the asynchronous transition within the time between the two latch’s strobe signals. The rate of failure is determined by the actual size of the sampling window of the data latch, and by the amount of time between the strobe signals of the two latches. Obviously, as the sampling window gets smaller, the number of times an asynchronous transition will occur during the sampling window will drop. In addition, however, a smaller sampling window is also indicative of a faster resolution time for an input transition which manages to fall within the sampling window.

B.2 80186 SYNCHRONIZERS

The 80186 contains synchronizers on the RES, TEST, TmrIn0–1, DRQ0–1, NMI, INTO–3, ARDY, and HOLD input lines. Each of these synchronizers use the two stage synchronization technique described above (with some minor modifications for the ARDY line, see section 3.1.6). The sampling window of the latch is designed to be in the tens of pico-seconds, and should allow operation of the synchronizers with a mean time between failures of over 30 years assuming continuous operation.
APPENDIX C
80186 EXAMPLE DMA INTERFACE CODE

$mod186
name assembly.example.80186.DMA.support
This file contains an example procedure which initializes the 80186 DMA
to perform the DMA transfers between the 80186 system and the
8272 Floppy Disk Controller (FDC). It assumes that the 80186
peripheral control block has not been moved from its reset location.

arg1 equ word ptr [BP + 4]
arg2 equ word ptr [BP + 6]
arg3 equ word ptr [BP + 8]
DMA.FROM.LOWER equ 0FFC0h ; DMA register locations
DMA.FROM.UPPER equ 0FFC2h
DMA.TO.LOWER equ 0FFC4h
DMA.TO.UPPER equ 0FFC6h
DMA.COUNT equ 0FFC8h
DMA.CONTROL equ 0FFC4h
DMA.TO.DISK.CONTROL equ 01460h
DMA.FROM.DISK.CONTROL equ 0A046h
FDC.DMA equ 6B8h
FDC.DATA equ 688h
FDC.STATUS equ 680h
cgroup group code
code segment
class public set.dma.
assume cs:cgroup

set.dma (offset,to) programs the DMA channel to point one side to the
disk DMA address, and the other to memory pointed to by ds:offset. If
'to' = 0 then will be a transfer from disk to memory; if
'to' = 1 then will be a transfer from memory to disk. The parameters to
the routine are passed on the stack.

set.dma proc near
enter 0,0 ; set stack addressability
push AX ; save registers used
push BX
push DX
test arg2,1 ; check to see direction of
je from.disk ; transfer

performing a transfer from memory to the disk controller

mov AX,DS ; get the segment value
rol AX,4 ; get the upper 4 bits of the
bits of the register

210973–84
mov BX,AX
mov DX,DMA,FROM.UPPER
out DX,AX
and AX,OFF0h
add AX,Arg1
mov DX,DMA,FROM.LOWER
out DX,AX
jnc no.carry.from
inc BX
mov AX,BX
mov DX,DMA,FROM.UPPER
out DX,AX

no.carry.from:
mov AX,FDC.DMA
mov DX,DMA.TO.LOWER
out DX,AX
xor AX,AX
mov DX,DMA.TO.UPPER
out DX,AX
mov AX,DMA.TO.DISK.CONTROL
mov DX,DMA.CONTROL
jnc no.carry.to
inc BX
mov AX,BX
mov DX,DMA.TO.UPPER
out DX,AX

no.carry.to:
mov AX,FDC.DMA
mov DX,DMA,FROM.LOWER
out DX,AX
xor AX,AX
mov DX,DMA,FROM.UPPER
out DX,AX
mov AX,DMA,FROM.DISK.CONTROL
mov DX,DMA.CONTROL
out DX,AX
pop DX
pop BX
pop AX
leave
ret

set.dma..
endp

code
ends

210973–85

210973–86

5-66
APPENDIX D
80186 EXAMPLE TIMER INTERFACE CODE

$mod186
name example.80186.timer.code

; this file contains example 80186 timer routines. The first routine
; sets up the timer and interrupt controller to cause the timer
; to generate an interrupt every 10 milliseconds, and to service
; interrupt to implement a real time clock. Timer 2 is used in
; this example because no input or output signals are required.
; The code example assumes that the peripheral control block has
; not been moved from its reset location (FFFF-F000 in I/O space).

arg1 equ word ptr [BP + 4]
arg2 equ word ptr [BP + 6]
arg3 equ word ptr [BP + 8]
timer.2int equ 19
; timer 2 has vector type 19

timer.2control equ 0FF66h

timer.2max.ctl equ 0FF64h

timer.int.ctl equ 0FF32h

col.register equ 0FF22h

interrupt.stat equ 0FF30h

data segment
public hour._minute..second..msec..
msec.. db ?
hour.. db ?
minute.. db ?
second.. db ?
data ends

cgroup group code
dgroup group data
code segment
public set.time.
assume cs:code,ds:dgroup

set.time(hour.minute.second) sets the time variables, initializes the
80186 timer2 to provide interrupts every 10 milliseconds, and
programs the interrupt vector for timer 2
set.time.

proc near
enter 0,0
push AX
push DX
push SI
push DS
xor AX,AX
mov DS,AX
mov SI,4 * timer2.int

210973–87
mov word ptr DS:[SI], offset timer_2_interrupt_routine
inc SI
inc SI
mov DS:[SI], CS
pop DS
mov AX, arg
mov hour, AL
mov AX, arg2
mov minute, AL
mov AX, arg3
mov second, AL
mov msec, 0
mov DX, timer2.maxctl
mov AX, 20000
out DX, AX
mov DX, timer2.control
mov AX, 111000000000001b
out DX, AX
mov DX, timer.intctl
mov AX, 0000b
out DX, AX
sti
endp

set.time.

timer2_interrupt_routine proc far
push AX
push DX

cmp msec, 99
jae bump.second
inc msec
jmp reset.intctl

bump.second:
mov msec, 0
cmp second, 59
jse bump.minute
inc second
jmp reset.intctl

bump.minute:
mov second, 0
cmp minute, 59
jse bump.hour
inc minute
jmp reset.intctl

pop DX
pop AX
ret

timer2_interrupt_routine endp

ends

210973–88

210973–89

5-68
bump.hour:
  mov minute.,0
  cmp hour.,12
  jae reset.hour
  inc hour.
  jmp reset.int.ctl
reset.hour:
  mov hour.,1
reset.int.ctl:
  mov DX,esi.register
  mov AX,8000h
  out DX,AX
  pop DX
  pop AX
  iret

$mod186
name example.80186.baud.code
:
:: this file contains example 80186 timer routines. The second routine
:: sets up the timer as a baud rate generator. In this mode,
:: Timer 1 is used to continually output pulses with a period of
:: 6.5 usec for use with a serial controller at 9600 baud
:: programmed in divide by 16 mode (the actual period required
:: for 9600 baud is 6.51 usec). This assumes that the 80186 is
:: running at 8 MHz. The code example also assumes that the
:: peripheral control block has not been moved from its reset
:: location (FF00-FFFF in I/O space).
:
timer1.control equ OFF5Eh
timer1.max.cnt equ OFF5Ah
code segment public 'code'
  assume cs:code
:: set.baud() initializes the 80186 timer1 as a baud rate generator for
:: a serial port running at 9600 baud
::
:: set.baud.
  proc near
  push AX
  push DX
  mov DX,timer1.max.cnt
  mov AX,13
  out DX,AX
  mov DX,timer1.control
  mov AX,1100000000000001b
  out DX,AX
  pop DX
  pop AX

210973-90
this file contains example 80186 timer routines. The third routine sets up the timer as an external event counter. In this mode, Timer 1 is used to count transitions on its input pin. After the timer has been set up by the routine, the number of events counted can be directly read from the timer count register at location FF58H in I/O space. The timer will count a maximum of 65535 timer events before wrapping around to zero. This code example also assumes that the peripheral control block has not been moved from its reset location (FF00-FFFF in I/O space).

```
;     timer1.control    equ OFF5Eh
; timer1.max.cnt      equ OFF5Ah
; timer1.cnt.reg      equ OFF58H

; code
; set.count() initializes the 80186 timer1 as an event counter

; set.count,       proc near
push AX
push DX
mov DX,timer1.max.cnt ; set the max count value
mov AX,0              ; allows the timer to count all the way to FFFF
out DX,AX
mov DX,timer1.control ; set the control word
mov AX,11000000000000101b
out DX,AX
xor AX,AX             ; zero AX
mov DX,timer1.cnt.reg ; and zero the count in the timer count register
out DX,AX
pop DX
pop AX
ret
```

; set.count,       endp
; code
; ends
; end
APPENDIX E

80186 EXAMPLE INTERRUPT CONTROLLER INTERFACE CODE

This routine configures the 80186 interrupt controller to provide two cascaded interrupt inputs (through an external 8259A interrupt controller on pins INT0/INT2) and two direct interrupt inputs (on pins INT1 and INT3). The default priority levels are used. Because of this, the priority level programmed into the control register is set to the 111, the level all interrupts are programmed to at reset.

```
$mod186
.name example.80186.interrupt.code

>This routine configures the 80186 interrupt controller to provide two cascaded interrupt inputs (through an external 8259A interrupt controller on pins INT0/INT2) and two direct interrupt inputs (on pins INT1 and INT3). The default priority levels are used. Because of this, the priority level programmed into the control register is set to the 111, the level all interrupts are programmed to at reset.

.int0.control equ OFF38H
.int.mask equ OFF28H

.code segment public 'code'

.set.int proc near

.push DX
.push AX

.mov AX,0100111B ; Cascade Mode: interrupt unmasked

.mov AX,01001101B ; now unmask the other external interrupts

.mov AX,0100110B ; Cascade Mode

.mov AX,0100111B ; interrupt unmasked

.endp

.code ends

.end

$mod186
.name example.80186.interrupt.code

>This routine configures the 80186 interrupt controller into Slave Mode. This code does not initialize any of the 80186 integrated peripheral control registers, nor does it initialize the external 8259A interrupt controller.

.relocation.reg equ OFFFEH

.code segment public 'code'

.assume CS:code

.push DX
.push AX

.mov AX,0100111B ; Cascade Mode

.mov AX,01001101B ; now unmask the other external interrupts

.mov AX,0100110B ; Cascade Mode

.mov AX,0100111B ; interrupt unmasked

.endp

.code ends

.end
```

This routine configures the 80186 interrupt controller into Slave Mode. This code does not initialize any of the 80186 integrated peripheral control registers, nor does it initialize the external 8259A interrupt controller.

```
$mod186
.name example.80186.interrupt.code

>This routine configures the 80186 interrupt controller into Slave Mode. This code does not initialize any of the 80186 integrated peripheral control registers, nor does it initialize the external 8259A interrupt controller.

.relocation.reg equ OFFFEH

.code segment public 'code'

.assume CS:code

.push DX
.push AX

.mov AX,0100111B ; Cascade Mode

.mov AX,01001101B ; now unmask the other external interrupts

.mov AX,0100110B ; Cascade Mode

.mov AX,0100111B ; interrupt unmasked

.endp

.code ends

.end
```

This routine configures the 80186 interrupt controller into Slave Mode. This code does not initialize any of the 80186 integrated peripheral control registers, nor does it initialize the external 8259A interrupt controller.

```
$mod186
.name example.80186.interrupt.code

>This routine configures the 80186 interrupt controller into Slave Mode. This code does not initialize any of the 80186 integrated peripheral control registers, nor does it initialize the external 8259A interrupt controller.

.relocation.reg equ OFFFEH

.code segment public 'code'

.assume CS:code

.push DX
.push AX

.mov AX,0100111B ; Cascade Mode

.mov AX,01001101B ; now unmask the other external interrupts

.mov AX,0100110B ; Cascade Mode

.mov AX,0100111B ; interrupt unmasked

.endp

.code ends

.end
```
APPENDIX F

80186/8086 EXAMPLE SYSTEM INITIALIZATION CODE

name example.80186.system.init

; This file contains a system initialization routine for the 80186
; or the 8086.  The code determines whether it is running on
; an 80186 or an 8086, and if it is running on an 80186, it
; initializes the integrated chip select registers.

restart segment at 0FFFFh

; This is the processor reset address at 0FFFF0H

org 0
jmp far ptr initialize
restart ends

init.hw segment at 0FF0h
assume CS:init.hw

; This segment initializes the chip selects. It must be located in the
; top 1K to ensure that the ROM remains selected in the 80186
; system until the proper size of the select area can be programmed.

UMCS.reg equ 0FFA0H
LMCS.reg equ 0FFA2H
PACS.reg equ 0FFA4H
MPCS.reg equ 0FFA8H
UMCS.value equ 0F038H
LMCS.value equ 0FF8H
PACS.value equ 007EH
MPCS.value equ 81B8H

initialize proc far

mov AX,2
mov CL,33
shr AX,CL
test AX,1
jz not.80186

mov DX,UMCS.reg
mov AX,UMCS.value
out DX,AX

mov DX,LMCS.reg
mov AX,LMCS.value
out DX,AX

mov DX,PACS.reg
mov AX,PACS.value
out DX,AX

mov DX,MPCS.reg
mov AX,MPCS.value
out DX,AX

; Now that the chip selects are all set up, the main program of the
; computer may be executed.

not.80186:
jmp far ptr monitor

initialize endp
init.hw ends
end

210973-93 210973-94

5-72
APPENDIX G

80186 WAIT STATE PERFORMANCE

Because the 80186 contains separate bus interface and execution units, the actual performance of the processor will not degrade at a constant rate as wait states are added to the memory cycle time from the processor. The actual rate of performance degradation will depend on the type and mix of instructions actually encountered in the user's program.

Shown below are two 80186 assembly language programs, and the actual execution time for the two programs as wait states are added to the memory system of the processor. These programs show the two extremes to which wait states will or will not affect system performance as wait states are introduced.

Program 1 is very memory intensive. It performs many memory reads and writes using the more extensive memory addressing modes of the processor (which also take a greater number of bytes in the opcode for the instruction). As a result, the execution unit must constantly wait for the bus interface unit to fetch and perform the memory cycles to allow it to continue. Thus, the execution time of this type of routine will grow quickly as wait states are added, since the execution time is almost totally limited to the speed at which the processor can run bus cycles.

Note also that this program execution time calculated by merely summing up the number of clock cycles given in the data sheet will typically be less than the actual number of clock cycles actually required to run the program. This is because the numbers quoted in the data sheet assume that the opcode bytes have been prefetched and reside in the 80186 prefetch queue for immediate access by the execution unit. If the execution unit cannot access the opcode bytes immediately upon request, dead clock cycles will be inserted in which the execution unit will remain idle, thus increasing the number of clock cycles required to complete execution of the program.

On the other hand, program 2 is more CPU intensive. It performs many integer multiplies, during which time the bus interface unit can fill up the instruction prefetch queue in parallel with the execution unit performing the multiply. In this program, the bus interface unit can perform bus operations faster than the execution unit actually requires them to be run. In this case, the performance degradation is much less as wait states are added to the memory interface. The execution time of this program is closer to the number of clock cycles calculated by adding the number of cycles per instruction because the execution unit does not have to wait for the bus interface unit to place an opcode byte in the prefetch queue as often. Thus, fewer clock cycles are wasted by the execution unit laying idle for want of instructions. Table G-1 lists the execution times measured for these two programs as wait states were introduced with the 80186 running at 8 MHz.

<table>
<thead>
<tr>
<th># of Waiting States</th>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exec Time (µsec)</td>
<td>Perf Degr</td>
</tr>
<tr>
<td>0</td>
<td>505</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>595</td>
<td>18%</td>
</tr>
<tr>
<td>2</td>
<td>669</td>
<td>12%</td>
</tr>
<tr>
<td>3</td>
<td>752</td>
<td>12%</td>
</tr>
</tbody>
</table>
table db 256 dup (?)
t.string db 64 dup (?)
m.array dw 32 dup (?)

data ends

code segment public 'code'
assume CS:egroup,DS:dgroup
public bench.1,bench.2,nothing,waitstate_set.timer.
bench.1 proc near
push SI
push CX
push BX
push AX

mov CX,64 ; translate 64 bytes
mov SI,0
mov BH,0

loop.back:

mov BL,t.string[SI] ; get the byte
mov AL,t.string[BX] ; translate byte
mov t.string[SI],AL ; and store it
inc SI ; increment index
loop loop.back ; do the next byte

pop AX
pop BX
pop CX
pop SI
ret

bench.1 endp

bench.2 proc near
push AX
push SI
push CX

mov CX,32 ; multiply 32 numbers
mov SI,offset m.array

loop.back.2:
imul AX,word ptr [SI],3 ; immediate multiply
mov word ptr [SI],AX
inc SI
inc SI
loop loop.back.2

pop CX
pop SI
pop AX
ret

bench.2 endp

210973-96

5-74
wait_state(n) sets the 80186 LMCS register to the number of wait states (0 to 3) indicated by the parameter n (which is passed on the stack). No other bits of the LMCS register are modified.

```
wait_state.
    proc near
    enter 0,0
    push AX
    push BX
    push DX
    mov BX,word ptr [BP + 4] ; get argument
    mov AX,OFFA2h ; get current LMCS register
    in AX,DX
    and AX,OFFFCCh ; and off existing ready bits
    and BX,3 ; insure ws count is good
    or AX,BX ; adjust the ready bits
    out DX,AX ; and write to LMCS
    pop DX
    pop BX
    pop AX
    leave ; tear down stack frame
    ret
endp
```

set.timer() initializes the 80186 timers to count microseconds. Timer 2 is set up as a prescaler to timer 0, the microsecond count can be read directly out of the timer 0 count register at location FF50H in I/O space.

```
set.timer.
    proc near
    push AX
    push DX
    mov DX,OFF66h ; stop timer 2
    mov AX,4000h
    out DX,AX
    mov AX,OFF50h ; clear timer 0 count
    out DX,AX
    mov AX,OFF52h ; timer 0 counts up to 65535
    out DX,AX
    ret
endp
```
<table>
<thead>
<tr>
<th>mov</th>
<th>DX,0ff56h; enable timer 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>AX,0x009h</td>
</tr>
<tr>
<td>out</td>
<td>DX,AX</td>
</tr>
<tr>
<td>mov</td>
<td>DX,0ff66h; clear timer 2 count</td>
</tr>
<tr>
<td>mov</td>
<td>AX,0</td>
</tr>
<tr>
<td>out</td>
<td>DX,AX</td>
</tr>
<tr>
<td>mov</td>
<td>DX,0ff62h; set maximum count of timer 2</td>
</tr>
<tr>
<td>mov</td>
<td>AX,2</td>
</tr>
<tr>
<td>out</td>
<td>DX,AX</td>
</tr>
<tr>
<td>mov</td>
<td>DX,0ff66h; re-enable timer 2</td>
</tr>
<tr>
<td>mov</td>
<td>AX,0x0001h</td>
</tr>
<tr>
<td>out</td>
<td>DX,AX</td>
</tr>
<tr>
<td>pop</td>
<td>DX</td>
</tr>
<tr>
<td>pop</td>
<td>AX</td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>set_timer</td>
<td>endp</td>
</tr>
<tr>
<td>code</td>
<td>ends</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX H
80186 NEW INSTRUCTIONS

The 80186 performs many additional instructions to those of the 8086. These instructions appear shaded in the instruction set summary at the back of the 80186 data sheet. This appendix explains the operation of these new instructions. In order to use these new instructions with the 8086/186 assembler, the "$mod186" switch must be given to the assembler. This can be done by placing the line: "$mod186" at the beginning of the assembly language file.

PUSH IMMEDIATE

This instruction allows immediate data to be pushed onto the processor stack. The data can be either an immediate byte or an immediate word. If the data is a byte, it will be sign extended to a word before it is pushed onto the stack (since all stack operations are word operations).

PUSHA, POPA

These instructions allow all of the general purpose 80186 registers to be saved on the stack, or restored from the stack. The registers saved by this instruction (in the order they are pushed onto the stack) are AX, CX, DX, BX, SP, BP, SI, and DI. The SP value pushed onto the stack is the value of the register before the first PUSH (AX) is performed; the value popped for the SP register is ignored.

This instruction does not save any of the segment registers (CS, DS, SS, ES), the instruction pointer (IP), the flag register, or any of the integrated peripheral registers.

IMUL BY AN IMMEDIATE VALUE

This instruction allows a value to be multiplied by an immediate value. The result of this operation is 16 bits long. One operand for this instruction is obtained using one of the 80186 addressing modes (meaning it can be in a register or in memory). The immediate value can be either a byte or a word, but will be sign extended if it is a byte. The 16-bit result of the multiplication can be placed in any of the 80186 general purpose or pointer registers.

This instruction requires three operands: the register in which the result is to be placed, the immediate value, and the second operand. Again, this second operand can be any of the 80186 general purpose registers or a specified memory location.

SHIFTS/ROTATES BY AN IMMEDIATE VALUE

The 80186 can perform multiple bit shifts or rotates where the number of bits to be shifted is specified by an immediate value. This is different from the 8086, where only a single bit shift can be performed, or a multiple shift can be performed where the number of bits to be shifted is specified in the CL register.

All of the shift/rotate instructions of the 80186 allow the number of bits shifted to be specified by an immediate value. Like all multiple bit shift operations performed by the 80186, the number of bits shifted is the number of bits specified modulus 32 (i.e., the maximum number of bits shifted by the 80186 multiple bit shifts is 31).

These instructions require two operands: the operand to be shifted (which may be a register or a memory location specified by any of the 80186 addressing modes) and the number of bits to be shifted.

BLOCK INPUT/OUTPUT

The 80186 adds two new input/output instructions: INS and OUTS. These instructions perform block input or output operations. They operate similarly to the string move instructions of the processor.

The INS instruction performs block input from an I/O port to memory. The I/O address is specified by the DX register; the memory location is pointed to by the DI register. After the operation is performed, the DI register is adjusted by 1 (if a byte input is specified) or by 2 (if a word input is specified). The adjustment is either an increment or a decrement, as determined by the Direction bit in the flag register of the processor. The ES segment register is used for memory addressing, and cannot be overridden. When preceded by a REPpeat prefix, this instruction allows blocks of data to be moved from an I/O address to a block of memory. Note that the I/O address in the DX register is not modified by this operation.
The OUTS instruction performs block output from memory to an I/O port. The I/O address is specified by the DX register; the memory location is pointed to by the SI register. After the operation is performed, the SI register is adjusted by 1 (if a byte output is specified) or by 2 (if a word output is specified). The adjustment is either an increment or a decrement, as determined by the Direction bit in the flag register of the processor. The DS segment register is used for memory addressing, but can be overridden by using a segment override prefix. When preceded by a REPpeat prefix, this instruction allows blocks of data to be moved from a block of memory to an I/O address. Again note that the I/O address in the DX register is not modified by this operation.

Like the string move instruction, these two instructions require two operands to specify whether word or byte operations are to take place. Additionally, this determination can be supplied by the mnemonic itself by adding a “B” or “W” to the basic mnemonic, for example:

```plaintext
INSB ;perform byte input
REP OUTSW ;perform word block output
```

BOUND

The 80186 supplies a BOUND instruction to facilitate bound checking of arrays. In this instruction, the calculated index into the array is placed in one of the general purpose registers of the 80186. Located in two adjacent word memory locations are the lower and upper bounds for the array index. The BOUND instruction compares the register contents to the memory locations, and if the value in the register is not between the values in the memory locations, an interrupt type 5 is generated. The comparisons performed are SIGNED comparisons. A register value equal to either the upper bound or the lower bound will not cause an interrupt.

This instruction requires two arguments: the register in which the calculated array index is placed, and the word memory location which contains the lower bound of the array (which can be specified by any of the 80186 memory addressing modes). The memory location containing the upper bound of the array must follow immediately the memory location containing the lower bound of the array.

ENTER AND LEAVE

The 80186 contains two instructions which are used to build and tear down stack frames of higher level, block structured languages. The instruction used to build these stack frames is the ENTER instruction. The algorithm for this instruction is:

```
PUSH BP /*save the previous frame pointer*/
if level=0 then
    BP:=SP;
else    templ:=SP;/*save current frame pointer*/
    temp2:= level - 1;
do while temp2>0 /*copy down previous level frame*/
    BP:= BP - 2; /*pointers*/
PUSH [BP];
BP:=templ;
PUSH BP; /*put current level frame pointer*/
/*in the save area*/
SP:=SP - disp; /*create space on the stack for*/
/*local variables*/
```
Figure H-1 shows the layout of the stack before and after this operation.

This instruction requires two operands: the first value (disp) specifies the number of bytes the local variables of this routine require. This is an unsigned value and can be as large as 65535. The second value (level) is an unsigned value which specifies the level of the procedure. It can be as great as 255.

The 80186 includes the LEAVE instruction to tear down stack frames built up by the ENTER instruction.

As can be seen from the layout of the stack left by the ENTER instruction, this involves only moving the contents of the BP register to the SP register, and popping the old BP value from the stack.

Neither the ENTER nor the LEAVE instructions save any of the 80186 general purpose registers. If they must be saved, this must be done in addition to the ENTER and the LEAVE. In addition, the LEAVE instruction does not perform a return from a subroutine. If this is desired, the LEAVE instruction must be explicitly followed by the RET instruction.
APPENDIX I
80186/80188 DIFFERENCES

The 80188 is exactly like the 80186, except it has an 8 bit external bus. It shares the same execution unit, timers, peripheral control block, interrupt controller, chip select, and DMA logic. The differences between the two caused by the narrower data bus are:

- The 80188 has a 4 byte prefetch queue, rather than the 6 byte prefetch queue present on the 80186. The reason for this is since the 80188 fetches opcodes one byte at a time, the number of bus cycles required to fill the smaller queue of the 80188 is actually greater than the number of bus cycles required to fill the queue of the 80186. As a result, a smaller queue is required to prevent an inordinate number of bus cycles being wasted by prefetching opcodes to be discarded during a jump.

- AD8–AD15 on the 80186 are transformed to A8–A15 on the 80188. Valid address information is present on these lines throughout the bus cycle of the 80188. Valid address information is not guaranteed on these lines during idle T states.

- BHE/S7 is always defined HIGH by the 80188, since the upper half of the data bus is non-existent.

- The DMA controller of the 80188 only performs byte transfers. The B/W bit in the DMA control word is ignored.

- Execution times for many memory access instructions are increased because the memory access must be funneled through a narrower data bus. The 80188 also will be more bus limited than the 80186 (that is, the execution unit will be required to wait for the opcode information to be fetched more often) because the data bus is narrower. The execution time within the processor, however, has not changed between the 80186 and 80188.

Another important point is that the 80188 internally is a 16-bit machine. This means that any access to the integrated peripheral registers of the 80188 will be done in 16-bit chunks, not in 8-bit chunks. All internal peripheral registers are still 16-bits wide, and only a single read or write is required to access the registers. When a word access is made to the internal registers, the BIU will run two bus cycles externally.

Access to the control block may also be done with byte operations. Internally the full 16-bits of the AX register will be written, while externally, only one bus cycle will be executed.
There are two operating modes of the 80C186 and 80C188: Compatible Mode and Enhanced Mode. In Compatible Mode, the 80C186 will function identically to the 80186 with the following noted exceptions:

1) All non-initialized registers in the peripheral control block will reset to a random value on power-up on the 80C186. Non-Initialized registers consist of those registers which are not used for control, i.e., address pointers, max count, etc. For compatibility, all registers should be programmed before being used on existing 80186 applications as well as on new 80C186 applications.

2) The ET (Esc/Trap) bit in the relocation register has no effect in Compatible Mode. If an escape opcode is executed, the 80C186 will always trap to an interrupt vector type 7. The 80C186 does not support any numerics operations when in Compatible Mode.

In Enhanced Mode, the 80C186 provides additional features not found on the 80186. There are newly defined registers to support these new features, and three of the output pins of the 80C186 change functionality. The new registers and pin descriptions are covered in Section 9.0.

The 80C188 in Enhanced Mode functions similarly to the 80C186 except for numerics operation. It is not possible to interface a numerics coprocessor with the 80C188. Therefore, none of the MCS pins change functionality when invoking Enhanced Mode on the 80C188. Further, any attempted execution of an escape opcode will result in a trap to interrupt vector type 7.
### 80C186 DESIGNS

<table>
<thead>
<tr>
<th>RAM Type</th>
<th>Row Address (A0–AX)</th>
<th>Column Address (A0–AX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K x 1</td>
<td>(128K Bytes) A1–A8</td>
<td>A9–A16</td>
</tr>
<tr>
<td>16K x 4</td>
<td>(32K Bytes) A1–A8</td>
<td>A9–A14</td>
</tr>
<tr>
<td>256K x 1</td>
<td>(512K Bytes) A1–A9</td>
<td>A9–A16</td>
</tr>
<tr>
<td>64K x 4</td>
<td>(128K Bytes) A1–A8</td>
<td>A9–A16</td>
</tr>
<tr>
<td>1M x 1</td>
<td>(2M Bytes) A1–A10</td>
<td>A11–A19 (+ Bank)</td>
</tr>
<tr>
<td>256K x 4</td>
<td>(512K Bytes) A1–A9</td>
<td>A10–A18</td>
</tr>
</tbody>
</table>

### 80C188 DESIGNS

**NOTE:** Address bit A0 can be used in either RAS or CAS addresses, so long as it is not included in any refresh address bits.

<table>
<thead>
<tr>
<th>RAM Type</th>
<th>Row Address (A0–AX)</th>
<th>Column Address (A0–AX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K x 1</td>
<td>(64K Bytes) A1–A7</td>
<td>A8–A15</td>
</tr>
<tr>
<td>16K x 4</td>
<td>(16K Bytes) A1–A7</td>
<td>A8–A13</td>
</tr>
<tr>
<td>256K x 1</td>
<td>(256K Bytes) A1–A8</td>
<td>A9–A17</td>
</tr>
<tr>
<td>64K x 4</td>
<td>(64K Bytes) A1–A8</td>
<td>A0, A9–A15</td>
</tr>
<tr>
<td>1M x 1</td>
<td>(1M Bytes) A1–A9</td>
<td>A10–A19</td>
</tr>
<tr>
<td>256K x 4</td>
<td>(256K Bytes) A1–A9</td>
<td>A0, A10–A17</td>
</tr>
</tbody>
</table>

### RAM Type | RAS Add | CAS Add | Refresh Add
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64K x 1</td>
<td>A0–A7</td>
<td>A0–A7</td>
<td>A0–A6</td>
</tr>
<tr>
<td>16K x 4</td>
<td>A0–A7</td>
<td>A0–A5</td>
<td>A0–A6</td>
</tr>
<tr>
<td>256K x 1</td>
<td>A0–A8</td>
<td>A0–A8</td>
<td>A0–A7</td>
</tr>
<tr>
<td>64K x 4</td>
<td>A0–A7</td>
<td>A0–A7</td>
<td>A0–A7</td>
</tr>
<tr>
<td>1M x 1</td>
<td>A0–A9</td>
<td>A0–A9</td>
<td>A0–A8</td>
</tr>
<tr>
<td>256K x 4</td>
<td>A0–A8</td>
<td>A0–A8</td>
<td>A0–A8</td>
</tr>
</tbody>
</table>
High Speed Numerics with the 80186/80188 and 8087

STEVE FARRER
APPLICATIONS ENGINEER
1.0 INTRODUCTION

From their introduction in 1982, the highly integrated 16-bit 80186 and its 8-bit external bus version, the 80188, have been ideal processor choices for high-performance, low-cost embedded control applications. The integrated peripheral functions and enhanced 8086 CPU of the 80186 and 80188 allow for an easy upgrade of older generation control applications to achieve higher performance while lowering the overall system cost through reduced board space, and a simplified production flow.

More and more controller applications need even higher performance in numerics, yet still require the low-cost and small form factor of the 80186 and 80188. The 8087 Numerics Data Coprocessor satisfies this need as an optional add-on component.

The 8087 Numeric Data Coprocessor is interfaced to the 80186 and 80188 through the 82188 IBC (Integrated Bus Controller). The IBC provides a highly integrated interface solution which replaces the 8288 used in 8086–8087 systems. The IBC incorporates all the necessary bus control for the 8087 while also providing the necessary logic to support the interface between the 80186/8 and the 8087.

This application note discusses the design considerations associated with using the 8087 Numeric Data Coprocessor with the 80186 and 80188. Sections two, three, and four contain an overview of the integrated circuits involved in the numerics configuration. Section five discusses the interfacing aspects between the 80186/8 and the 8087, including the role of the 82188 Integrated Bus Controller and the operation of the integrated peripherals on the 80186/8 with the 8087. Section six compares the advantages of using an 8087 Numeric Data Coprocessor over software routines written for the host processor as well as the advantage of using an 80186/8 numerics system over an 8086/8088 numerics system.

Except where noted, all future references to the 80186 will apply equally to the 80188.

2.0 OVERVIEW OF THE 80186

The 80186 and 80188 are highly integrated microprocessors which effectively combine up to 20 of the most common system components onto a single chip. The 80186 and 80188 processors are designed to provide both higher performance and a more highly integrated solution to the total system.

Higher integration results from integrating system peripherals onto the microprocessor. The peripherals consist of a clock generator, an interrupt controller, a DMA controller, a counter/timer unit, a programmable wait state generator, programmable chip selects, and a bus controller. (See Figure 1.)

![Figure 1. 80186/8 Block Diagram](231590-1)
Higher performance results from enhancements to both general and specific areas of the 8086 CPU, including faster effective address calculation, improvement in the execution speed of many instructions, and the inclusion of new instructions which are designed to produce optimum 80186 code.

The 80186 and 80188 are completely object code compatible with the 8086 and 8088. They have the same basic register set, memory organization, and addressing modes. The differences between the 80186 and 80188 are the same as the differences between the 8086 and 8088: the 80186 has a 16-bit architecture and 16-bit bus interface; the 80188 has a 16-bit internal architecture and an 8-bit data bus interface. The instruction execution times of the two processors differ accordingly: for each non-immediate 16-bit data read/write instruction, 4 additional clock cycles are required by the 80188.

3.0 NUMERICS OVERVIEW

3.1 The Benefits of Numeric Coprocessing

The 8086/8 and 80186/8 are general purpose microprocessors, designed for a very wide range of applications. Typically, these applications need fast, efficient data movement and general purpose control instructions. Arithmetic on data values tends to be simple in these applications. The 8086/8 and 80186/8 fulfill these needs in a low cost, effective manner.

However, some applications require extremely fast and complex math functions which are not provided by a general purpose processor. Such functions as square root, sine, cosine, and logarithms are not directly available in a general purpose processor. Software routines required to implement these functions tend to be slow and not very accurate. Integer data types and their arithmetic operations (i.e., add, subtract, multiply and divide) which are directly available on general purpose processors, still may not meet the needs for accuracy, speed and ease of use.

Providing fast, accurate, complex math can be quite complicated, requiring large areas of silicon on integrated circuits. A general data processor does not provide these features due to the extra cost burden that less complex general applications must take on. For such features, a special numeric data processor is required — one which is easy to use and has a high level of support in hardware and software.

3.2 Introduction to the 8087

The 8087 is a numeric data coprocessor which is capable of performing complex mathematical functions while the host processor (i.e. the main CPU) performs more general tasks. It supports the necessary data types and operations and allows use of all the current hardware and software support for the 8086/8 and 80186/8 microprocessors. The fact that the 8087 is a coprocessor means it is capable of operating in parallel with the host CPU, which greatly improves the processing power of the system.

The 8087 can increase the performance of floating-point calculations by 50 to 100 times, providing the performance and precision required for small business and graphics applications as well as scientific data processing.

The 8087 numeric coprocessor adds 68 floating-point instructions and eight 80-bit floating-point registers to the basic 8086 programming architecture. All the numeric instructions and data types of the 8087 are used by the programmer in the same manner as the general data types and instructions of the host.

The numeric data formats and arithmetic operations provided by the 8087 support the proposed IEEE Microprocessor Floating Point Standard. All of the proposed IEEE floating point standard algorithms, exception detection, exception handling, infinity arithmetic and rounding controls are implemented. The IEEE standard makes it easier to use floating point and helps to avoid common problems that are inherent to floating point.

3.3 Escape Instructions

The coprocessing capabilities of the 8087 are achieved by monitoring the local bus of the host processor. Certain instructions within the 8086 assembly language known as ESCAPE instructions are defined to be coprocessor instructions and, as such, are treated differently.

The coprocessor monitors program execution of the host processor to detect the occurrence of an ESCAPE instruction. The fetching of instructions is monitored via the data bus and bus cycle status S2-S0, while the execution of instructions is monitored via the queue status lines QS0 and QS1.

All ESCAPE instructions start with the high-order 5-bits of the instruction opcode being 11011. They have two basic forms, the memory reference form and the non-memory reference form. The non-memory form, shown in Figure 2A, initiates some activity in the coprocessor using the nine available bits of the ESCAPE instruction to indicate which function to perform.

Memory reference forms of the ESCAPE instruction, shown in Figure 2B, allow the host to point out a memory operand to the coprocessor using any host memory.
addressing mode. Six bits are available in the memory reference form to identify what to do with the memory operand.

Memory reference forms of ESCAPE instructions are identified by bits 7 and 6 of the byte following the ESCAPE opcode. These two bits are the MOD field of the 8086/8 or 80186/8 effective address calculation byte. Together with the R/M field (bits 2 through 0), they determine the addressing mode and how many subsequent bytes remain in the instruction.

### 3.4 Host Response to Escape Instructions

The host performs one of two possible actions when encountering an ESCAPE instruction: do nothing (operation is internal to 8087) or calculate an effective address and read a word value beginning at that address (required for all LOADS and STORES). The host ignores the value of the word read and hence the cycle is referred to as a “Dummy Read Cycle.” ESCAPE instructions do not change any registers in the host other than advancing the IP. If there is no coprocessor or the coprocessor ignores the ESCAPE instruction, the ESCAPE instruction is effectively a NOP to the host. Otherwise calculating a memory address and reading a word of memory, the host makes no other assumptions regarding coprocessor activity.

The memory reference ESCAPE instructions have two purposes: to identify a memory operand and, for certain instructions, to transfer a word from memory to the coprocessor.

### 3.5 Coprocessor Response to Escape Instructions

The 8087 performs basically three types of functions when encountering an ESCAPE instruction: LOAD (read from memory), STORE (write to memory), and EXECUTE (perform one of the internal 8087 math functions).

When the host executes a memory reference ESCAPE instruction intended to cause a read operation by the 8087, the host always reads the low-order word of any 8087 memory operand. The 8087 will save the address and data read. To read any subsequent words of the operand, the 8087 must become a local bus master.

When the 8087 has the local bus, it increments the 20-bit physical address it saved to address the remaining words of the operand.

When the ESCAPE instruction is intended to cause a write operation by the 8087, the 8087 will save the address but ignore the data read. Eventually, it will get control of the local bus and perform successive writes incrementing the 20-bit address after each word until the entire numeric variable has been written.

ESCAPE instructions intended to cause the execution of a coprocessor calculation do not require any bus activity. Numeric calculations work off of an internal register stack which has been initialized using a LOAD operation. The calculation takes place using one or two of the stack positions specified by the ESCAPE instruction. The result of the operation is also placed in one of the stack positions specified by the ESCAPE instruction. The result may then be returned to memory using a STORE instruction, thus allowing the host processor to access it.
4.0 OVERVIEW OF THE 82188 INTEGRATED BUS CONTROLLER

4.1 Introduction

The 82188 Integrated Bus Controller (IBC) is a highly integrated version of the 8288 Bus Controller. The IBC provides command and control timing signals for bus control and all of the necessary logic to interface the 80186 to the 8087.

4.2 Bus Control Signals

The bus command and control signals consist of RD, WR, DEN, DT/R, and ALE. The timings and levels are driven following the latching of valid signals on the status lines S0–S2. When S0–S2 change state from passive to active, the IBC begins cycling through a state machine which drives the corresponding control and command lines for the bus cycle. As with the 8288, an address enable input (AEN) is present to allow tri-stat-

5.0 DESIGNING THE SYSTEM

5.1 Circuit Schematics of the 80186/8–82188–8087 System

![Figure 3. 80186/8–82188–8087 Circuit Diagram](231590-2)

Figure 3. 80186/8–82188–8087 Circuit Diagram

5-87
5.2 Queue Status

The 8087 tracks the instruction execution of the 80186 by keeping an internal instruction queue which is identical to the processor's instruction queue. Each time the processor performs an instruction fetch, the 8087 latches the instruction into its own queue in parallel with the processor. Each time the processor removes the first byte of an instruction from the queue, the 8087 removes the byte at the top of the 8087 queue and checks to see if the byte is an ESCAPE prefix. If it is, the 8087 decodes the following bytes in parallel with the processor to determine which numeric instruction the bytes represent. If the first byte of the instruction is not an ESCAPE prefix, the 8087 discards it along with the subsequent bytes of the non-numeric instruction as the 80186 removes them from the queue for execution.

The 8087 operates its internal instruction queue by monitoring the two queue status lines from the CPU. This status information is made available by the CPU by placing it into queue status mode. This requires strapping the RD pin on the 80186 around. When RD is tied to ground, ALE and WR become QS0 (Queue Status #0) and QS1 (Queue Status #1) respectively.

Each time the 80186 begins decoding a new instruction, the queue status lines indicate "first byte of instruction taken from queue". This signals the 8087 to check for an ESCAPE prefix. As the remaining bytes of the instruction are removed, the queue status indicates "subsequent byte removed from queue". The 8087 uses this status to either continue decoding subsequent bytes, if the first byte was an ESCAPE prefix, or to discard the subsequent bytes if the first byte was not an ESCAPE prefix.

The QS0(ALE) and QS1(WR) pins of the 80186 are fed directly to the 82188 where they are latched and delayed by one-half-clock. The delayed queue status from the 82188 is then presented directly to the 8087.

The waveforms of the queue status signals are shown in Figure 4. The critical timings are the setup time into the 82188 from the 80186 and the setup and hold time into the 8087 from the 82188. The calculations for an 8 MHz system are as follows:

\[
\begin{align*}
.5T_{CLCL} - T_{CHQS} (186 \text{ max}) & \geq T_{QVCL} (82188 \text{ min}) \geq 15 \text{ ns} \\
T_{CLCL} - T_{CLQOV} (82188 \text{ max}) & \geq T_{QVCL} \geq 10 \text{ ns} \\
T_{CLQOV} (82188 \text{ min}) & \geq T_{CLQX} (8087 \text{ min}) \geq 5 \text{ ns}
\end{align*}
\]

![Figure 4. Queue Status Timing](231590-3)

<table>
<thead>
<tr>
<th>Table 1. Queue Status Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>QS1</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

---

5-88
5.3 Bus Control Signals

When the 80186 is in Queue Status mode, another component must generate the ALE, RD, and WR signals. The 82188 provides these signals by monitoring the CPU bus cycle status (S0–S2). Also provided are DEN and DT/R which may be used for extra drive capability on the control bus. With the exception of ALE, all control signals on the 82188 are almost identical to their corresponding 80186 control signals. This section discusses the differences between the 80186 and the 82188 control signals for the purpose of upgrading an 80186 design to an 80186–8087 design. For original 80186–8087 designs, there is no need to compare control signal timings of the 82188 with the 80186.

5.3.1 ALE

The ALE (Address Latch Enable) signal goes active one clock phase earlier on the 80186 than on the 82188. Timing of the ALE signal on the 82188 is closer to that of the 8086 and 8288 bus controller because the bus cycle status is used to generate the ALE pulse. ALE on the 80186 goes active before the bus cycle status lines are valid.

The inactive edge of ALE occurs in the same clock phase for both the 80186 and the 82188. The setup and hold times of the 80186 address relative to the 82188 ALE signal are shown in Figure 5 and are calculated for an 8 MHz system as follows:

Setup Time
For 80186 = \( T_{AVCH} (186 \text{ min}) + T_{CHLL} (82188 \text{ min}) \)
\[ = 10 + 0 = 10 \text{ ns.} \]

For 8087 = 0.5 \((T_{CLCL} - T_{CLAV} (8087 \text{ max}) + T_{CHLL} (82188 \text{ min})) \)
\[ = 0.5 (125) - 55 + 0 = 7.5 \]

Hold Time
\[ = 0.5 (T_{CLCL}) - T_{CHLL} (82188 \text{ max}) + T_{CLAZ} (186 \text{ min}) \]
\[ = 0.5 (125) - 30 + 10 = 42.5 \text{ ns.} \]
5.3.2 Read and Write

The read and write signals of the 82188 have identical timings to those of the 80186 with one exception: the 82188 WR inactive edge may not go inactive quite as early as the 80186. This spec is, in fact, a tighter spec than the 80186 WR timing and should make designs easier. The timings for RD and WR are shown in Figure 6 for both the 80186 and the 82188.

5.3.3 DEN

The DEN signal on the 82188 is identical to the DEN signal on the 80186 but with a tighter timing specification. This makes designs easier with the 82188 and makes upgrades from 80186 bus control to 82188 bus control more straightforward. The timings for DEN on both the 80186 and 82188 are shown in Figure 7.

5.3.4 DT/R

The operation of the DT/R signal varies somewhat between the 80186 and the 82188. The 80186 DT/R signal will remain in an active high state for all write cycles and will default to a high state when the system bus is idle (i.e., no bus activity). The 80186 DT/R goes low only for read cycles and does so only for the duration of the bus cycle. At the end of the read cycle, assuming the following cycle is a non-read, the DT/R signal will default back to a high state. Back-to-back read cycles will result in the DT/R signal remaining low until the end of the last read cycle.

The DT/R signal on the 82188 operates differently by making transitions only at the start of a bus cycle. The 82188 DT/R signal has no default state and therefore will remain in whichever state the previous bus cycle required. The 82188 DT/R signal will only change states when the current bus cycle requires a state different from the previous bus cycle.
5.4 Chip Selects

5.4.1 Introduction

Chip-select circuitry is typically accomplished by using a discrete decoder to decode two or more of the upper address lines. When a valid address appears on the address bus, the decoder generates a valid chip-select. With this method, any bus master capable of placing an address on the system bus is able to generate a chip-select. An example of this is shown in Figure 9 where an 8086/8087 system uses a common decoder on the address bus. Note the decoder is able to operate regardless of which processor is in control of the bus.

With high integration processors like the 80186 and 80188, the chip-select decoder is integrated onto the processor chip. The integrated chip-selects on the 80186 enable direct processor connection to the chip-enable pins on many memory devices, thus eliminating an external decoder. But because the integrated chip-selects decode the 80186's internal bus, an external bus master, such as the 8087, is unable to activate them. The 82188 IBC solves this problem by supplying a chip-select mechanism which may be activated by both the host processor and a second processor.

5.4.2 CSI and CSO of the 82188

The CSI (chip select in) and CSO (chip select out) pins of the 82188 provide a way for a second bus master to select memory while also making use of the 80186 integrated chip-selects. The CSI pin of the 82188 connects directly to one of the 80186's chip-selects while CSO connects to the memory device designated for the chip-selects range. An example of this is shown in Figure 10.
When the 80186 has control of the bus, the circuit acts just as a buffer and the memory device gets selected as if the circuit had not been there. Whenever CSI goes active, CSO goes active. When a second bus master, such as the 8087, takes control of the bus, CSO goes active and remains active until the 8087 passes control back to the processor. At this time CSO is deactivated.

A functional block diagram of the CSI–CSO circuit is shown in Figure 11. A grant pulse on the RQ/GT0 line gives control to the 8087 and also causes the 8087CONTROL signal to go active, which in turn causes CSO to go active. The 8087CONTROL signal goes inactive when either a release is received on RQ/GT0, indicating that the 8087 is relinquishing control to the main processor, or a grant is received on the RQ/GT1 line, indicating that the 8087 is relinquishing control to a third processor. Both actions signify that the 8087 is relinquishing the bus. If CSO goes inactive because a third processor took control of the bus, then CSO will go active again for the 8087 when a release pulse is transmitted on the RQ/GT1 line to the 8087. This release pulse occurs as a result of SYSHLDA going inactive from the third processor.

5.4.3 SYSTEM DESIGN EXAMPLE

To provide the 8087 access to data in low memory through an integrated chip-select, the LCS pin should be disconnected from the bank that it is currently selecting and fed directly into the 82188 CSI. The CSI output should be connected to the banks which the LCS formerly selected. The LCS will still select the same banks because CSO goes active whenever CSI goes active. But now the 8087, when taking control of the bus, may also select these banks.

Care must be taken in locating the 8087 data area because it must reside in the area in which the chip-select is defined. If the 8087 generates an address outside of the LCS range, the CSO will still go active, but the address will erroneously select a part of the lower bank. Note also that this chip-select limits the size of the 8087 data area to the maximum size memory which can be selected with one chip-select. However, this does not place a limit on instruction code size or non-8087 data size. All 80186 and 8087 instructions are fetched by the processor and therefore do not require that the 8087 be

![Figure 11. 82188 Chip Select Circuitry](image-url)
able to address them. Likewise, non-8087 data is never accessed by the 8087 and therefore does not require an 8087 chip-select.

5.5 Wait State & Ready Logic

The 8087 must accurately track every instruction fetch the 80186 performs so that each op-code may be read from the system bus by the 8087 in parallel with the processor. This means that for instruction code areas, the 80186 cannot use internally generated wait states. All ready logic for these areas must be generated externally and sent into the 82188. The 82188 then presents a synchronous ready out (SRO) signal to both the 80186 and the 8087.

5.5.1 INTERNAL WAIT STATES WITH INSTRUCTION FETCHES

If internal wait states are used by the processor with the 8087 at zero wait states, then the 8087 will latch op-codes using a four clock bus cycle while the processor is using between five and seven clocks on each bus cycle. If the wait states are truly necessary to latch valid data from memory, then a four clock bus cycle will force the 8087 to latch invalid data. The invalid data may then be possibly interpreted to be an ESCAPE prefix when, in reality, it is not. The reverse may also hold true in that the 8087 may not recognize an ESCAPE prefix when it is fetched. These conditions could cause a system to hang (i.e., cease to operate), or operate with erroneous results.

If the memory is fast enough to allow latching of valid data within a four clock bus cycle, then the 80186 internal wait states will not cause the system to hang. Both processors will receive valid data during their respective bus cycles. The 8087 will finish its bus cycle earlier than the processor, but this is of no consequence to system operation. The 8087 will synchronize with the processor using the status lines S0–S2 at the start of the next instruction fetch.

5.5.2 INTERNAL WAIT STATES WITH DATA & I/O CYCLES

With the exception of “Dummy Read Cycles” and instruction fetches, all memory and I/O bus cycles executed by the host processor are ignored by the 8087. Coprocessor synchronization is not required for untracked bus cycles and, therefore, internally generated wait states do not affect system operation. All of the I/O space and any part of memory used strictly for data may use the internal wait state generator on the 80186.

Memory used for 8087 data is somewhat different. Here, as in the case of code segment areas, the system must rely on an external ready signal or else the memory must be fast enough to support zero wait state operation. Without an external ready signal, the 8087 will always perform a four clock bus cycle which, when used with slow memories, results in the latching of invalid data.

Internal wait states will not affect system operation for data cycles performed by the 8087. In this case the 8087 has control of the bus and the two processors operate independently.

One type of data cycle has not yet been considered. Each time a numerics variable is accessed, the host processor runs a “Dummy Read Cycle” in order to calculate the operand address for the 8087. The 8087 latches the address and then takes control of the bus to fetch any subsequent bytes which are necessary. If the 8087 variables are located at even addresses, then an internally generated wait state will not present any problems to the system. If any numeric variables are located at odd addresses, then the interface between the host and coprocessor becomes asynchronous causing erroneous results.

The erroneous results are due to the 80186 running two back-to-back bus cycles with wait states while the 8087 runs two back-to-back bus cycles without wait states. The start of the second bus cycle is completely uncoordinated between the two processors and the 8087 is unable to latch the correct address for subsequent transfers. For this reason, 8087 variables in a 80186 system must always lie on even boundaries when using the internal wait state generator to access them.

Numeric variables in an 80188 system must never be in a section of memory which uses the internal wait state generator. The 80188 will always perform consecutive bus cycles which would be equivalent to the 80186 performing an odd addressed “Dummy Read Cycle.”

5.5.3 AUTOMATIC WAIT STATES AT RESET

The 80186 automatically inserts three wait states to the predefined upper memory chip select range upon power up and reset. This enables designers to use slow memories for system boot ROM if so desired. If slow ROM’s are chosen, then no further programming is necessary. If fast ROM’s are chosen, then the wait state logic may simply be reprogrammed to the appropriate number of wait states.

The automatic wait states have the possibility of presenting the same problem as described in section 5.5.1 if
the boot ROM needs one or more wait states. Under these conditions the 8087 would be forced to latch invalid opcodes and possibly mistake one for an ESCAPE instruction.

If the boot ROM requires wait states, then some sort of external ready logic is necessary. This allows both processors to run with the same number of wait states and insures that they always receive valid data.

If the boot ROM does not require wait states, then there is no need to design external ready logic for the upper chip select region. But if 8087 code is present in the upper memory chip select region, the situation described in section 3.4 regarding "Dummy Read Cycles" must be considered.

The 82188 solves this problem by inserting three wait states on the SRO line to the 8087 for the first 256 bus cycles. By doing this the 82188 inserts the same number of wait states to both processors keeping them synchronized. The initialization code for the 80186 must program the upper memory chip select to look at external ready and to insert zero wait states within these first 256 bus cycles. At the end of the 256 bus cycles, the 82188 stops inserting wait states and both processors run at zero wait states.

5.5.4 EXTERNAL READY SYNCHRONIZATION

The 80186 and 8087 sample READY on different clock edges. This implies that some sort of external synchronization is required to insure that both processors sample the same ready state. Without the synchronization, it would be possible for the external signal to change state between samples. The 80186 may sample ready high while the 8087 samples ready low. This would lead to the two processors running different length bus cycles and possibly cause the system to hang.

The 82188 provides ready synchronization through the ARDY and SRDY inputs. Once a valid transition is recorded, the 82188 presents the results on the SRO output and holds the output in that state until both processors have had a chance to sample the signal.

5.6 BUS ARBITRATION

In order for the 8087 to read and write numeric data to and from memory, it must have a means of taking control of the local bus. With the 8086/88 this is accomplished through a request-grant exchange protocol. The 80186, however, makes use of HOLD/HOLD AC-KNOWLEDGE protocol to exchange control of the bus with another processor. The 82188 supplies the necessary conversion to interface RQ/GT to HOLD/HLDA signals. The RQ/GT signal of the 8087 connects directly to the 82188's RQ/GT0 input while the 82188's HOLD and HLDA pins connect to the 80186's HOLD and HLDA pins.

When the 8087 requires control of the bus, the 8087 sends a request on the RQ/GT0 line to the 82188. The 82188 responds by sending a HOLD request to the 80186. When HLDA is received back from the 80186, the 82188 sends a grant back to the 8087 on the same RQ/GT0 line.

The 82188 also has provisions for adding a third bus-master to the system which uses HOLD/HLDA protocol. This is accomplished by using the 82188 SYSHOLD, SYSHLDA, and RQ/GT1 signals. The third processor requests the bus by pulling the SYSHOLD line high. The 82188 will route (and translate if necessary) the requests to the current bus master. If the 8087 has control, the 82188 will request control via the RQ/GT1 line which should be connected to the 8087's RQ/GT1 line.

The 8087 will relinquish control by getting off the bus and sending a grant pulse on the RQ/GT1 line. The 82188 responds by sending a SYSHLDA to the third processor. The third processor lowers SYSHOLD when it has finished on the bus. The 82188 routes this in the form of a release pulse on the RQ/GT1 line to the 8087. The 8087 then continues bus activity where it left off. The maximum latency from SYSHOLD to SYSHLDA is equal to the 80186 latency + 8087 latency + 82188 latency.

5.7 SPEED REQUIREMENTS

One of the most important timing specs associated with the 80186-8087 interface is the speed at which the system should run. The 8087 was designed to operate with a 33% duty cycle clock whereas the 80186 and 80188 were designed to operate with a 50% duty cycle clock. In order to run both parts off the same clock, the 8087 must run at a slower speed than is typically implied by its dash number in the 8086/88 family.
To determine the speed at which an 8087 may run (with a 50% duty cycle clock), the minimum low and high times of the 8087 must be examined. The maximum of these two minimum specs becomes the half-period of the 50% duty cycle system clock. For example, the 8087-1 provides worst case spec compatibility with the 80186 at system clock-speeds of up to 8 MHz. The clock waveforms are shown in Figure 12 using 10 MHz timings.

The minimum clock low time (TCLCH) of the 10 MHz 8087 is 53 ns. The clock low time of an 8 MHz 80186 is specified to be:

$$\frac{1}{4}(T_{CLCH}) - 7.5$$

Solving for TCLCL of the 80186 using TCLCH of the 8087 yields the following:

$$\frac{1}{4}(T_{CLCL}) - 7.5 = T_{CLCH}$$

$$T_{CLCL} = 2(T_{CLCH} + 7.5)$$

$$T_{CLCL} = 121 \text{ ns}$$

The calculation shows minimum cycle time of the 80186 to be 121 ns. This time translates into a maximum frequency of 8.26 MHz.

### 6.0 BENCHMARKS

#### 6.1 Introduction

The following benchmarks compare the overall system performance of an 8086, 80188, and an 80186 in numeric applications. Results are shown for all three processors in systems with the 8087 coprocessor and in systems using an 8087 software emulator. Three FORTRAN benchmark programs are used to demonstrate the large increase in floating-point math performance provided by the 8087 and also the increase in performance due to the enhanced 80186 and 80188 host processors.

The 8086 results were measured on an Intellec® Series III Microcomputer Development System with an iSBC® 86/12 board and an iSBC 337 multimodule. Typically, one wait state for memory read cycles and two wait states for memory write cycles are experienced in this environment.

The 80186 and 80188 results were measured on a prototype board which allowed zero wait state operation at 8 MHz. The benchmarks measured using the 8087 showed little sensitivity to wait states. Instructions executed on the 8087 tend to be long in comparison to the amount of bus activity required and, therefore, are not affected much by wait states.

The benchmarks measured using the software emulator are much more bus intensive and average from 10 to 15 percent performance degradation for one wait state.

All execution times shown here represent 8 MHz operation. The 8086 results were measured at 5 MHz and extrapolated to achieve 8 MHz execution times.

#### 6.2 Interest Rate Calculations

Routines were written in FORTRAN-86 to calculate the final value of a fund given the annual interest and the present value. It is assumed that the interest will be compounded daily, which requires the calculation of the yearly effective rate. This value, which is the equivalent annual interest if the interest were compounded daily, is determined by the following formula:

$$yer = (1 + (ir/np))^{*np} - 1$$
where:
- yer is the yearly effective rate
- ir is the annual interest rate
- np is the number of compounding periods per annum

Once the yer is determined, the final value of the fund is determined by the formula:

\[ fv = (1 + \text{yer}) \times \text{pv} \]

where:
- pv is the present value
- fv is the future value

Results are obtained using single-precision, double-precision, and temporary real precision operands when:
- ir is set to 10% (0.1)
- np is set to 365 (for daily compounding)
- pv is set to $2,000,000

### THE RESULTS:

<table>
<thead>
<tr>
<th></th>
<th>yer</th>
<th>Final Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Precision (32-bit)</td>
<td>10.514%</td>
<td>$2,210,287.50</td>
</tr>
<tr>
<td>Double-Precision (64-bit)</td>
<td>10.516%</td>
<td>$2,210,311.57</td>
</tr>
<tr>
<td>Temporary Real Precision</td>
<td>10.516%</td>
<td>$2,210,311.57</td>
</tr>
</tbody>
</table>

The difference between the final single-precision and double-precision values is $24.07; the difference in the final value between the double-precision and the temporary real precision is 0.000062 cents. Since the 8087 performs all internal calculations on 80-bit floating point numbers (temp real format), temporary real precision operations perform faster than single- or double-precision. No data conversions are required when loading or storing temporary real values in the 8087. Thus, for business applications, the double-precision computing of the 8087 is essential for accurate results, and the performance advantage of using the 8087 turns out to be as much as 100 times the equivalent software emulation program.

### 6.3 Matrix Multiply Benchmark Routine

A routine was written in FORTRAN-86 to compute the product of two matrices using a simple row/column inner-product method. Execution times were obtained for the multiplication of 32 × 32 matrices using double precision. The results of the benchmark are shown in Figure 14.

The results show the 8087 coprocessor systems performing from 23 to 31 times faster than the equivalent software emulation program. Both the 80188/87 and the 80186/87 systems outperform the 8086/87 system by 34 to 75 percent. This difference is mainly attributed to the fact that the matrix program largely consists of effective address calculations used in array accessing. The hardware effective address calculator of the 80186 and 80188 allow each array access to improve by as much as three times the 8086 effective address calculation.

### 6.4 Whetstone Benchmark Routine

The Whetstone benchmark program was developed by Harry Curnow for the Central Computer Agency of the British government. This benchmark has received high visibility in the scientific community as a measurement of main frame computer performance. It is a "synthetic" program. That is, it does not solve a real problem, but rather contains a mix of FORTRAN statements which reflect the frequency of such statements as measured in over 900 actual programs. The program computes a performance metric: "thousands of Whetstone instructions per second (KIPS)."

Simple variable and array addressing, fixed- and floating-point arithmetic, subroutine calls and parameter passing, and standard mathematical functions are performed in eleven separate modules or loops of a prescribed number of iterations.

### Table 2. Interest Rate Benchmark Results

<table>
<thead>
<tr>
<th></th>
<th>8087 Software Emulator</th>
<th>8087 Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>70.3 ms</td>
<td>62.8 ms</td>
</tr>
<tr>
<td>Double Precision</td>
<td>72.1 ms</td>
<td>62.9 ms</td>
</tr>
<tr>
<td>Temp Real Precision</td>
<td>72.6 ms</td>
<td>63.0 ms</td>
</tr>
<tr>
<td>Average</td>
<td>71.7 ms</td>
<td>62.9 ms</td>
</tr>
</tbody>
</table>
The original coding of the Whetstone benchmark was written in Algol-60 and used single-precision values. It was rewritten in FORTRAN with single-precision values to exactly reflect the original intent. Another version was created using double-precision values. The results are shown in Table 3.

- The results show the 8087 systems with the 80186 and 80188 outperforming the equivalent software emulation by 60 to 83 times. Additionally, the 80186 coupled with the 8087 outperformed the 8086/87 system by 22 percent.
Table 3. Whetstone Benchmark Results

<table>
<thead>
<tr>
<th>Units = KIPS</th>
<th>8087 Software Emulator</th>
<th>8087 Coprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>80188</td>
<td>80186</td>
</tr>
<tr>
<td>Single Precision</td>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td>Double Precision</td>
<td>2</td>
<td>2.2</td>
</tr>
</tbody>
</table>

6.5 Benchmark Conclusions

These benchmarks show that the 8087 Numeric Data Coprocessor, coupled with either the 80186 or the 80188, can increase the performance of a numeric application by 75 to 100 times the equivalent software emulation program.

Applications which require array accessing with effective address calculations will benefit even more by using the 80188 and 80186 as the host processor as compared to the 8086. The results of the matrix multiplication show both the 80188 and 80186 outperforming the 8086 by 34 and 75%, respectively, in an 8087 system. In general, an 80186/8087 system will offer a 10% to a 75% improvement over an equivalent 8086/8087 system, depending on the instruction mix.

7. CONCLUSION

For controller applications which require high performance in numerics and low system cost, the 16-bit 80186 or 8-bit 80188 coupled with the 8087 offers an ideal solution. The integrated features of the 80186 and 80188 offer a low system cost through reduced board space and a simplified production flow while the 8087 fulfills the performance requirements of numeric applications.

The 82188 IBC provides a straightforward, highly integrated solution to interfacing the 80188 or 80186 to the 8087. The bus control timings of the 82188 are compatible with the 80186 and 80188, allowing easy upgrades from existing designs. The 82188 features present a highly integrated solution to both new and old designs.

The coprocessing capabilities of the 8087 bring performance improvements of 75 to 100 times the equivalent 80186 or 80188 software emulation program and an 80186/8087 system will offer a 10% to a 75% improvement over an equivalent 8086/8087 system depending on the instruction mix.

In addition a growing base of high-level language support (FORTRAN, Pascal, C, Basic, PL/M, etc.) from Intel and numerous third-party software vendors facilitates the timely and efficient generation of application software.

REFERENCES:

- 82188 Data Sheet #231051
- 80186 Data Sheet #210451
- 80188 Data Sheet #210706
- iAPX 86/88 80186/188 Users Manual
  - Programmers Reference #210911
  - Hardware Reference #210912
- AP-113 "Getting Started with the Numeric Data Processor #207865

5-98
80186/188 Interface to Intel Microcontrollers

PARVIZ KHODADADI
APPLICATIONS ENGINEER
1.0 INTRODUCTION

Systems which require I/O processing and serial data transmission are very software intensive. The communication task and I/O operations consume a lot of the system's intelligence and software. In many conventional systems the central processing unit carries the burden of all the communication and I/O operations in addition to its main routines, resulting in a slow and inefficient system.

In an ideal system, tasks are divided among processors to increase performance and achieve flexibility. One attractive solution is the combination of the Intel highly integrated 80186 microprocessor and the Intel 8-bit microcontrollers such as the 80C51, 8052, or 8044. In such a system, the 80186 provides the processing power and the 1 Mbyte memory addressability, while the controller provides the intelligence for the I/O operations and data communication tasks. The 80186 runs application programs, performs the high level communication tasks, and provides the human interface. The microcontroller performs 8-bit math and single bit boolean operations, the low level communication tasks, and I/O processing.

This application note describes an efficient method of interfacing the 16-bit 80186 high integration microprocessor to the 80C51, 8052, or the microcontroller-based 8044 serial communication controller. The interface hardware shown in Figure 1.1, is very simple and may be implemented with a programmable logic device or a gate-array. The 80186 and the microcontroller may run asynchronously and at different speeds. With this technique data transfers up to 200 Kbytes per second can be achieved between a 12 MHz microcontroller and an 8 MHz 80186.

The 8-bit 80188 high integration microprocessor can also be used with the same interface technique. The performance of the interface is the same since an 8-bit bus is used.

Interface to the 8044, 80C51, and the 8052 is identical because they have identical pinouts (some pins have alternate functions). As an example, the software procedures for the 8044/80186 interface, which is the building block for the application driver, is supplied in this Application Note.
1.1 System Overview

The 80186 and the microcontrollers are processors. They each access memory and have address/data, read, and write signals. There are three common ways to interface multiple processors together:

1) First In First Out (FIFO)
2) Dual Port RAM (DPRAM)
3) Slave Port

The FIFO interface, compared to DPRAM, requires less TTL and is easier to interface; however, FIFOs are expensive. The DPRAM interface is also expensive and even more complex. When DPRAM is used, the address/data lines of each processor must be buffered, and hardware logic is needed to arbitrate access to DPRAM. The slave port interface given here is cheaper and easier than both FIFO and DPRAM alternatives.

The 80186 processor, when interfaced to this circuit, views the microcontroller as a peripheral chip with 8-bit data bus and no address lines (see Figure 1.1). It can read status and send commands to the microcontroller at any time. The microcontroller becomes a slave coprocessor while keeping its processing power and serial communication capabilities.

The microcontrollers, with the interface hardware, have a high level command interface like many other data communication peripherals. For example, the 80186 can send the microcontroller commands such as Transmit or Configure. This means the designer does not have to write low level software to perform these tasks, and it offloads the 80186 to serve other functions in the application.

1.2 Application Examples:

The combination of the 80186 and a microcontroller basically provides all the functions that are needed in a system: a 16-bit CPU, 8-bit CPU, DMA controller, I/O ports, and a serial port. The 80C51 and the 8052 have an on-chip asynchronous channel, while the 8044 has an intelligent SDLC serial channel. In addition, many other functions such as timers, counters, and interrupt controllers are integrated in both the 80186 and the microcontrollers.

Applications of the system described above are in the area of robotics, data communication networks, or serial communication backplanes. A typical example is copiers. Different segments of the copy machine like the motor, paper feed, diagnostics, and error/warning displays are all controlled by microcontrollers. Each segment receives orders from and replies to the central processor which consists of the 80186 interfaced with a microcontroller.

Another common application is in the area of process controllers. An example is a central control unit for a multiple story building which controls the heating, cooling, and lighting of each room in each floor. In each room a microcontroller performs the above functions based on the orders received from the central processor. Depending on the throughput and type of the serial communication required, the 8044 or the 80C51 (8052) may be selected for the application.

2.0 OVERVIEW OF THE 80186, 80C51, 8052, AND 8044

This section briefly discusses the features of the microcontrollers and the 80186. For more information about these products please refer to the Intel Microcontroller and Microsystem components hand-books. Readers familiar with the above products may skip this section.

2.1 The 80186 Internal Architecture

The 80186 contains an enhanced version of Intel's popular 8086 CPU integrated with many other features common to most systems (Figure 2.1). The 16-bit CPU can access up to 1 Mbyte of memory and execute instructions faster than the 8086. With speed selection of 8, 10, and 12.5 MHz, this highly integrated product is the most popular 16-bit microprocessor for embedded control applications.

The on-chip DMA controller has two channels which can each be shared by multiple devices. Each channel is capable of transferring data up to 3.12 Mbytes per second (12.5 MHz speed). It offers the choice of byte or word transfer. It can be programmed to perform a burst transfer of a block of data, transfer data per specified time interval, or transfer data per external request.

The on-chip interrupt controller responds to both external interrupts and interrupts requested by the on-chip peripherals such as the timers and the DMA channels. It can be configured to generate interrupt vector addresses internally like the microcontrollers or externally like the popular 8259 interrupt controller. It can be configured to be a slave controller to an external interrupt controller (iRMX 86 mode) or be master for one or two 8259s which in turn may be masters for up to 8 more 8259s. When configured in master mode, each channel can support up to 64 external interrupts (128 total).

Three 16-bit timers are also integrated on the chip. Timer 0 and timer 1 can be configured to be 16-bit counters and count external events. If configured as timers, they can be started by software or by an external event. Timer 0 and 1 each contain a timer output pin. Transitions on these pins occur when the timers reach one of the two possible maximum counts. Timer
2 can be used as a prescaler for timer 0 and 1 or can be used to generate DMA requests to the on-chip DMA channel.

Finally, the integrated clock generator, the wait state generator, and the chip select logic reduce the external logic necessary to build a processing system.

### 2.2 The MCS-51 Internal Architecture

The 80C51BH, as shown in Figure 2.2, consists of an 8-bit CPU which can access up to 64 Kbytes of data memory (RAM) and 64 Kbytes of program memory (ROM). In addition, 4 Kbytes of ROM and 128 bytes of RAM are built onto the chip.

The on-chip interrupt controller supports five interrupts with two priority levels. There are two timers integrated in the 80C51. Timer 0 and 1 can be configured as 8-bit or 16-bit timers or event counters.

Finally, the integrated full duplex asynchronous serial channel provides the human interface or communication capability with other microcontrollers. The UART supports data rates up to 500 kHz (with 15 MHz crystal) and can distinguish between address bytes and data bytes.

The 8052 has the same features as the 80C51 except it has 8 Kbytes of on-chip ROM and 256 bytes of on-chip RAM. In addition, the 8052 has another timer which may be configured as the baud rate generator for the serial port.

### 2.3 The 8044 Internal Architecture

The 8044 has all the features of the 80C51. In addition the on-chip RAM size is increased to 192 bytes and an intelligent HDLC/SDLC serial channel (SIU) replaces the 80C51 serial port (see Figure 2.3). It supports data rates up to 2.4 Mbps when an external clock is used and 375 Kbps when the clock is extracted from the data line. The serial port can be used in half duplex point to point, multipoint, or one-way loop configurations.

---

**Figure 2.1. 80186 Block Diagram**

**Figure 2.2. 80C51 Block Diagram**

**Figure 2.3. 8044 Block Diagram**

**Figure 2.4. 8044 Automatic Response to SDLC Commands**
The SIU is called an intelligent channel because it responds to some SDLC commands automatically without the CPU intervention when it is set in auto mode. These automatic responses substantially reduce the communication software. Figure 2.4 gives the commands and the automatic responses.

The 8044 supports many types of frames including the standard SDLC format. Figure 2.5 shows the types of frames the 8044 can transmit and receive. If a format with an address byte is chosen, the 8044 performs address filtering during reception and transmits the contents of the station address register during transmission automatically. If a format with FCS bytes is chosen, the 8044 performs Cyclic Redundancy Check (CRC) during reception and calculates the FCS bytes during transmission of a frame in hardware. Two preamble bytes (PFS) may optionally be added to the frames. Formats that include the station address and the control byte are supported both in the auto and flexible modes.

### 3.0 80186/MICROCONTROLLER INTERACTION

The 80186 communicates with the microcontroller (8044, 80C51 or 8052) through the system's memory and the Command/Data and Status registers. The CPU creates a data structure in the memory, programs the DMA controller with the start address and byte count of the block, and issues a command to the microcontroller. A hypothetical block diagram of a microcontroller when used with the interface hardware is given in Figure 3.1.

Chip select and interrupt lines are used to communicate between the microcontroller and the host. The interrupt is used by the microcontroller to draw the 80186's attention. The Chip Select is used by the 80186 to draw the microcontroller's attention to a new command.

There are two kinds of transfers over the bus: Command/Status and data transfers. Command/Status transfers are always performed by the CPU. Data transfers are requested by the microcontroller and are typically performed by the DMA controller.

The CPU writes commands using CS and WR signals and interrupts the microcontroller. The microcontroller reads the command, decodes it and performs the necessary actions. The CPU reads the status register using CS and RD signals (see Figure 4.1).

To initiate a command like TRANSmit or CONFIGure, a write operation to the microcontroller is issued by the CPU. A read operation from the CPU gives the status of the microcontroller. Section 5 discusses details on these commands and the status.

Any parameters or data associated with the command are transferred between the system memory and the microcontroller using DMA. The 80186 prepares a data block in memory. Its first byte specifies the length of the rest of the block. The rest of the block is the information field. The CPU programs the DMA controller with the start address of the block, length of the block and other control information and then issues the command to the microcontroller.

When the microcontroller requires access to the memory for parameter or data transfer, it activates the 80186 DMA request line and uses the DMA controller to achieve the data transfer. Upon completion of an operation, the microcontroller interrupts the 80186. The CPU then reads results of the operation and status of the microcontroller.
4.0 SYSTEM INTERFACE

There are two kinds of transfers over the bus: command/status and data transfers. The command/status transfers are always initiated and performed by the 80186. The data transfers are requested by the microcontroller using the DMA request (DRQ) line. In relatively slow systems the 80186 might also perform the data transfers. In that case, the request from the microcontroller will serve as an interrupt to the CPU. This mode of operation depends on the serial data rate.

The system interface performs command/status transfers, data/parameter transfers, and interrupts. This section describes the interface between the 80186 and a microcontroller shown in Figure 1.1. Section 6 describes the interface hardware.

4.1 Command/Status Transfers

The 80186 controls the microcontroller by writing into the command/data register and reading from the status register. The CPU writes a command by activating the chip select (PCS0), putting the command onto the data bus, and activating the WR signal. The command byte is latched into the command/data register, and the microcontroller is interrupted. In the interrupt service routine, the microcontroller reads the command byte from the command/data register, decodes the command byte, and activates the DRQ for data or parameter transfer if the decoded command requires such transfer.

At the end of parameter transfer the microcontroller updates the status register and interrupts the 80186.

4.2 Data/Parameter Transfer

Data/parameter transfers are controlled by a pair of REQUEST/ACKNOWLEDGE lines: DMA Request line (DRQ) and DMA Acknowledge line (DACK). Data and parameters are transferred via the Command/Data register to or from memory.

In order to request a transfer from memory, the microcontroller activates the DRQ pin. The DRQ signal goes active after a read operation by the microcontroller. In response, the 80186 DMA controller performs a byte transfer from the memory to the Command/Data register. Data is transferred on the bus and written into the Command/Data register on the rising edge of the 80186 WR signal (MWR), which is activated by the DMA controller. Figure 4.2 shows the write timing.

In order to request a transfer to memory, the microcontroller activates the DRQ signal and outputs the data into the Command/Data latch. When the microcontroller WR signal goes active, DRQ is set. In response, the DMA performs the data transfer and resets the DRQ signal. Figure 4.3 shows the read timing.
4.3 Interrupt

The microcontroller reports on completion of an event by updating the status register and raising the interrupt signal assuming this signal is initially low. The interrupt is cleared by the command from the CPU where the INTERRUPT ACKNOWLEDGE bit is set (MD7). The INTA bit is the most significant bit of the command byte. Figure 4.4 and 4.5 show the interrupt timing. Note that it is the responsibility of the CPU to clear the interrupt in order to prevent a deadlock.

<table>
<thead>
<tr>
<th>80186 Pin Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>RD</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DACK</td>
<td>RD</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**NOTE:**
Only one of CS, DACK may be active at any time.

**Figure 4.1. Data Bus Control Signals and Their Functions**

![Figure 4.1. Data Bus Control Signals and Their Functions](image1)

**Figure 4.2. Write Timing**

![Figure 4.2. Write Timing](image2)

**Figure 4.3. Read Timing**

![Figure 4.3. Read Timing](image3)
5.0 COMMANDS AND STATUS

This section specifies the format of the commands and status. The commands and status given here are similar to most common coprocessors and data communication peripherals (e.g., the 82588 and 82586). The user may add more commands or redefine the formats for his/her own specific application.

5.1 Commands

A command is given to the microcontroller by writing it into the Command/Data register and interrupting the microcontroller. The command can be issued at any time; but in case it is not accepted, the operation is treated like a NOP and will be ignored (although the INT will be updated).

Format:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>OPERATION</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.1.1 ACKNOWLEDGING INTERRUPT (BIT 7)

The INTA bit, if set, causes the interrupt hardware signal and the interrupt bit to be cleared. This is the only way to clear the interrupt bit and reset the 80186 interrupt signal other than by a hardware reset.

5.1.2 OPERATIONS (BITS 0–3)

The OPERATION field initiates a specific operation. The microcontroller executes the following commands in software:

- NOP
- ABORT
- TRANSMIT*
- CONFIGURE*
- DUMP*
- RECEIVE*
- TRA-DISABLE
- REC-DISABLE
*Requires DMA operation.

The above operations except ABORT are executed only when the microcontroller is not executing any other operation. Abort is accepted only when the CPU is performing a DMA operation.
Operations that require parameter transfer (e.g., CONFIGURE and DUMP) or data transfer (e.g., TRANSMIT and RECEIVE) are called parametric operations. The remaining are called non-parametric operations.

An operation is initiated by writing into the command register. This causes the microcontroller to execute the command decode instructions. Some of the operations cause the microcontroller to read parameters from memory. The parameters are organized in a block that starts with an 8-bit byte count. The byte count specifies the length of the rest of the block. Before beginning the operation, the DMA pointer of the DMA channel must point to the byte count. There is no restriction on the memory structure of the parameter block as long as the microcontroller receives the next byte of the block for every DMA request it generates. Transferring the bytes is the job of the 80186 DMA controller.

The microcontroller requests the byte-count and determines the length of the parameter block. It then requests the parameters.

Upon completion of the operation, (when interrupt is low) the microcontroller updates the status, raises the interrupt signal, and goes idle.

NOP
This operation does not affect the microcontroller. It has no parameters and no results.

ABORT
This operation attempts to abort the completion of an operation under execution. It is valid for CONFIGURE, TRANSMIT, DUMP, and RECEIVE. It is ignored for any of the above if transfer of parameters has already been accomplished. The microcontroller, upon reception of the ABORT command, stops the DMA operation and issues an Execution-Aborted interrupt.

TRANSMIT
This operation transmits one message. A message may be transmitted as an SDLC frame by the 8044, or in ASYNC protocol by the 80C51 or the 8052 serial port.

Figure 5.1 shows the format of the Transmit block. A typical transmit operation parameter block includes the destination address and the control byte in the information field. As an example, see the 8044 transmit block in Figure 7.2.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE COUNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRST INFO BYTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST INFO BYTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.1. Format of Transmit Block

The transmit operation will either complete the execution or be aborted by a specific ABORT operation. A Transmit-Done or Execution-Aborted interrupt is issued upon completion of this operation.

CONFIGURE
This operation configures the microcontroller’s internal registers. The length and the part of the configuration block that is modified are determined by the first two bytes of the command parameter (see Figure 5.2). The FIRST BYTE specifies the first register in the configuration block that will be configured, and the BYTE COUNT specifies the number of registers that will be configured starting with the FIRST BYTE. For example, if the FIRST BYTE is 1 and the BYTE COUNT is the length of the configure block, then all of the registers are updated. If FIRST BYTE is 4 and BYTE COUNT is 2, then only the fourth register in the configure block is updated. Minimum byte count is 2.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE COUNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRST BYTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRST REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST REGISTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2. Format of Configure Block

A Configure-Done interrupt is issued when the operation is done unless ABORT was issued during the DMA operation.

DUMP
This operation causes dumping of a set of microcontroller internal registers to system memory. Figure 7.4 shows the format of the 8044 DUMP block.

The DUMP operation will either complete the execution or be aborted by a specific ABORT operation. A Dump-Done or Execution-Aborted interrupt is issued upon completion of this operation.
RECEIVE

This operation enables the reception of frames. It is ignored if the microcontroller’s serial channel is already in reception mode.

The serial port receives only frames that pass the address filtering. The microcontroller transfers the received information and the byte count to the system memory using DMA. The completion of frame reception causes a Receive-Done event.

REC-DISABLE

This operation causes reception to be disabled. If transfer of data to the 80186 memory has already begun, then it is treated like the ABORT command. This operation has no parameters. REC-DISABLE is accepted only when the microcontroller’s serial port is in receive mode.

TRA-DISABLE

This operation causes the transmission process to be aborted. If the microcontroller is fetching data from 80186 memory, then it is treated like the ABORT command. This operation has no parameters. It is accepted only when the serial port is in transmit mode.

5.1.3 ILLEGAL COMMANDS

Parametric and non-parametric commands except ABORT will be rejected (interrupt will not be set) if the microcontroller is already executing a command.

ABORT is rejected if issued when the microcontroller is not requesting DMA operation, or a non-Parametric execution is performed, or transfer of parameters/data has already been accomplished.

DMA operations shall not be aborted by any non-parametric or parametric command except by the ABORT command.

REC-DISABLE and TRA-DISABLE will not be accepted if the serial channel is idle.

5.2 Status

The microcontroller provides the information about the last operation that was executed, via the status register.

The microcontroller reports on these events by updating a status register and raising the INTERRUPT signal. Information from the status register is valid provided the interrupt signal is high or bit 0 of the status being read is set.

Format:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTS*</td>
<td>RTS*</td>
<td>E</td>
<td>EVENT</td>
<td>DMA</td>
<td>INT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*8044 only

5.2.1 INTERRUPT (BIT 0)

The interrupt bit is set together with the hardware interrupt signal. Setting the INT bit indicates the occurrence of an event. This bit is cleared by any command whose INTA bit is set. Status is valid only when this bit is set.

5.2.2 DMA OPERATION (BIT 1)

The DMA bit, when set, indicates that a DMA operation is in progress. This bit is set if the command received by the microcontroller requires data or parameter transfer. If this bit is clear, DRQ will be inactive. The DMA bit, when cleared, indicates the completion of a DMA operation.

5.2.3 ERROR (BIT 5)

The E bit, if set, indicates that the event generated for the operation that was completed contains a warning, or the operation was not accepted.

5.2.4 REQUEST TO SEND (BIT 6)

The RTS bit, if clear, indicates that the serial channel is requesting a transmission.

5.2.5 CLEAR TO SEND (BIT 7)

The CTS bit indicates that, if the RTS bit is clear, the serial port is active and transmitting a frame.

5.2.6 EVENT (BITS 2–4)

The event field specifies why the microcontroller needs the attention of the 80186.

The following events may occur:

CONFIGURE-DONE
TRANSMIT-DONE
DUMP-DONE
RECEIVE-DONE
RECEPTION-DISABLE
TRANSMISSION-DISABLE
EXECUTION-ABORTED
CONFIGURE-DONE

This event indicates the completion of a CONFIGURE operation.

TRANSMIT-DONE

This event indicates the completion of the TRANSMIT operation.

If the E bit is set, it indicates that the transmit buffer was already full.

DUMP-DONE

This event indicates that the DUMP operation is completed.

RECEIVE-DONE

This event indicates that a frame has been received and stored in memory.

The format of the received message is indicated in Figure 5.3.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST INFO BYTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST INFO BYTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECEIVED BYTE COUNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3. Format of Receive Block

Following the byte count, a few more bytes relating to the received frame such as the source address and the control byte may be transferred to the system memory using DMA. As an example, see the 8044 receive block in Figure 7.3.

Note that the format of a frame received by the microcontroller serial channel is configured by the CONFIGURE command.

If the E bit is set, buffer overrun has occurred.

RECEPTION-DISABLED

This event is issued as a result of a RCV-DISABLE operation that causes part of a frame to be disabled.

If the E bit is set, the serial port was already disabled, and the RCV-DISABLE is not accepted.

TRANSMISSION-DISABLED

This event is issued as a result of a TRA-DISABLE operation that causes transmission of a frame to be disabled.

The E bit, if set, indicates that the TRA-DISABLE operation was not accepted since the serial port was already idle, or transmission of a frame has already been accomplished.

EXECUTION-ABORTED

This event indicates that the execution of the last operation was aborted by the ABORT command.

If the E bit is set, ABORT was issued when the microcontroller was not executing any commands.

6.0 HARDWARE DESCRIPTION

The interface hardware shown in Figures 6.1 and 6.2 are identical. The difference is the status register. In Figure 6.2, an external latch is used to latch the status byte. This hardware is recommended if an extra I/O port on the microcontroller is required for some other applications, or external program and data memory is required for the microcontroller. The hardware shown in Figure 6.1 makes use of one of the microcontroller’s I/O ports (Port 1) to latch the status to minimize hardware. The discussion of Sections 1 through 5 apply to both schematics.

6.1 Reset

After an 80186 hardware reset, the microcontroller is also reset. The on-chip registers are initialized as explained in the Intel Microcontroller Handbook. The reset signal also clears the 80186 interrupt and the microcontroller interrupt signals by resetting FF3 (Flip-Flop 3) and FF2 (Flip-Flop 2). Figure 4.5 shows the RESET timing.

6.2 Sending Commands

A bidirectional latched transceiver (74ALS646) is used for the Command/Data register. When the 80186 writes a command to the Command/Data register, it interrupts the microcontroller. The interrupt is generated only when bit 7 (INTA) of the command byte is set. When the 80186 PCS0 and WR signals go active to write the command, FF2 will be set and FF3 will be cleared. The output of FF3 is the interrupt to the 80186 and the INT status bit. The INT bit is cleared immediately to indicate that the status is no longer valid. The output of FF2 is the interrupt to the microcontroller. A high to low transition on this line will interrupt the microcontroller. The interrupt signal will be cleared as soon as the microcontroller reads the command from the Command/Data register.
6.3 DMA Transfers

In the interrupt service routine the command is decoded. If it requires a DMA transfer, the microcontroller sets the DMA bit of the status register which activates the DMA request signal. DRQ active causes the 80186 on-chip DMA to perform a fetch and a deposit bus cycle. The first DMA cycle clears the DRQ signal (FF1 is cleared). When the microcontroller performs a read or write operation, the output of the FF1 will be set, and DRQ goes active again.

The DMA controller transfers a byte from system memory to the Command/Data register. Data is latched when the 80186 PCS1 and WR signals go active. PCS1 and WR active also clear FF1. The microcontroller monitors the output of FF1 by polling the P3.3 pin. When FF1 is cleared the microcontroller reads the byte from the Command/Data register. The P3.3 pin is also the interrupt pin. If a slow rate of transfer is acceptable, every DMA transfer can be interrupt driven to allow the microcontroller to perform other tasks.

The DMA controller transfers a byte from the Command/Data register to system memory by activating the 80186 PCS1 and RD signals. PCS1 and RD active also clear FF1. When FF1 is cleared the microcontroller writes the next byte to the Command/Data register.

When all the data is transferred, the microcontroller clears the DMA status bit to disable DRQ. It then updates the status, sets the INT bit, and interrupts the 80186.

If the interface hardware in Figure 6.1 is used P1.1 is the DMA status bit and P1.0 is the INT bit. The microcontroller enables or disables them by writing to port 1. In Figure 6.2, DRQ or INT is disabled or enabled by writing to the 74LS374 status register. Note that the INT status bit is cleared by the hardware when the 80186 writes a command.

6.4 Reading Status

The command is written and the status is read with the same chip select (PCS0), although the status is read through the 74LS245 transceiver and the command is written to the Command/Data register.

---

Figure 6.1. Hardware Interface (Port 1 is the Status Register)
The microcontroller updates the status byte whenever a change occurs in the status and outputs the result to the status register. In order to read status, the 80186 activates the PCS0 line, and then activates the RD line. The contents of the status are put on the data bus, through the 74LS245 transceiver.

For systems that require two DMA channels, a second pair of DRQ1/DACK1 signals may easily be added to the hardware. In that case one of the status bits (DMA2) ANDed with the output of FF1 will serve as the second DMA request signal (DRQ1). DACK1 can be generated with the 80186 PCS2.

### 7.0 8044/80186 INTERFACE

This section shows how to make use of the status and commands described in section 5 and the hardware given in Figure 6.1 to interface the 80186 with the 8044. The 8044 code to implement these functions is shown in Appendix A.

### 7.1 Configuring the 8044

This operation configures the 8044 registers. The format of the configure block is shown in Figure 7.1. The part of the configuration block that is modified is determined by the first two bytes of the command parameter. The FIRST BYTE specifies the first register in the configure block that will be configured, and the BYTE COUNT specifies the number of registers that will be configured starting with the FIRST BYTE. For example, if the FIRST BYTE is 1 and the BYTE COUNT is 13, then all of the registers are updated. If FIRST BYTE is 4 and BYTE COUNT is 2, then transmit buffer start register is configured.

The configure command performs the following: 1) configures the interrupts and assigns their priorities; 2) assigns the start address and length of the transmit and receive buffers; 3) sets the station address; 4) sets the clock option and the frame format.
For other microcontrollers the format of the configure block should be modified accordingly. For example, the 80C51 serial port registers (e.g., T2CON, SCON) replace the 8044 SIU registers in the configure block.

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE COUNT</td>
<td>FIRST BYTE</td>
<td>STS</td>
<td>SMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATION ADDRESS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSMIT BUFFER ADDRESS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSMIT BUFFER LENGTH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECEIVE BUFFER START</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECEIVE BUFFER LENGTH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERRUPT PRIORITY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERRUPT ENABLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER/COUNTER MODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMER/COUNTER MODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROCESSOR STATUS WORD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 7.1. Format of the 8044 Configure Block

### 7.2 Transmitting a Message with the 8044

A message is a block of data which represents a text file or a set of instructions for a remote node or an application program which resides on the 8044 program memory. A message can be a frame (packet) by itself or can be comprised of multiple frames. An SDLC frame is the smallest block of data that the 8044 transmits. The 8044 can receive commands from the 80186 to transmit and receive messages. The 8044 on-chip CPU can be programmed to divide messages into frames if necessary. Maximum frame size is limited by the transmit or receive buffer.

To transmit a message, the 80186 prepares a transmit data block in memory as shown in Figure 7.2. Its first byte specifies the length of the rest of the block. The next two bytes specify the destination address of the node the message is being sent to and the control byte of the message. The 80186 programs the DMA controller with the start address of the block, length of the block and other control information and then issues the Transmit command to the 8044.

Upon receiving the command, the 8044 fetches the first byte of the block using DMA to determine the length of the rest of the block. It then fetches the destination address and the control byte using DMA.

The 8044 fetches the rest of the message into the on-chip transmit buffer. The size and location of the transmit buffer in the on-chip RAM is configured with the Configure command. The 8044 CPU then enables the Serial Interface Unit (SIU) to transmit the data as an SDLC frame. The SIU sends out the opening flag, the station address, the SDLC control byte, and the contents of transmit buffer. It then transmits the calculated CRC bytes and the closing flag. The 8044 CPU and the SIU operate concurrently. The CPU can fetch bytes from system memory or execute a command such as TRANSMIT-DISABLE while the SIU is active.

Upon completion of transmission, the SIU updates the internal registers and interrupts the 8044 CPU. The 8044 then updates the status and interrupts the 80186. Note that baud rate generation, zero bit insertion, NRZI encoding, and CRC calculation are automatically done by the SIU.

### 7.3 Receiving a Message with the 8044

To receive a message, the 80186 allocates a block of memory to store the message. It sets the DMA channel and sends the Receive command to the 8044.

Upon reception of the command, the 8044 enables its serial channel. The 8044 receives and passes to memory all frames whose address matches the individual or broadcast address and passes the CRC test.

The SIU performs NRZI decoding and zero bit deletion, then stores the information field of the received frame in the on-chip receive buffer. At the end of reception, the CPU requests the transfer of data bytes to 80186 memory using DMA. After transferring all the bytes, the 8044 transfers the data length, source address, and control byte of the received frame to the memory (see Figure 7.3). Upon completion of the transfers, the 8044 updates the status register and raises the interrupt signal to inform the 80186.

If the SIU is not ready when the first byte of the frame arrives, then the whole frame is ignored. Disabling reception after the first byte was passed to memory causes the rest of the frame to be ignored and an interrupt with Receive-Aborted event to be issued.
7.4 Dumping the 8044 Registers

Upon reception of the Dump command, the 8044 transfers the contents of its internal registers to the system memory (See Figure 7.4).

7.5 Aborting an Operation

To abort a DMA operation, the 80186 sends an Abort command to the Command/Data latch and interrupts the 8044. During a DMA operation, the 8044 puts the external interrupt to high priority; therefore, the Abort interrupt will suspend the execution of the operation in progress and update the status register with the Execution-Aborted event. It then returns the 8044 program counter to a location before the aborted operation started. The Abort software procedure given in Appendix A gives the details of the execution of the ABORT command.

7.6 Disabling the Transmission or Reception

Transmission of a frame is aborted if the 80186 sends a TRANSMIT-DISABLE command to the 8044. The command causes the 8044 to clear the Transmit Buffer.
Full (TBF) bit. During transmission, if the TBF bit is cleared, the SIU will discontinue the transmission and interrupt the 8044 CPU.

The RECEIVE-DISABLE command causes the 8044 to clear the Receive Buffer Empty (RBE) bit. The SIU aborts the reception, if the RBE bit is cleared by the CPU.

When transmission or reception of a frame is discontinued, the SIU interrupts the 8044 CPU. The CPU then updates the status and interrupts the 80186.

### 7.7 Handling Interrupts

When the 80186 sends a command, it sets the 8044 external interrupt flag. The 8044 services the interrupt at its own convenience. In the interrupt service routine the 8044 executes the appropriate instructions for a given command. During execution of a command the 8044 ignores any command, except ABORT, sent by the 80186 (see section 5.1.2). This is accomplished by clearing the interrupt flag before the 8044 returns from the interrupt service routine. During DMA operations the 8044 sets the external interrupt to high priority. An interrupt with high priority can suspend execution of an interrupt service routine with low priority. The ABORT command given by the 80186 will interrupt the execution of the DMA transfer in progress. Upon completion of ABORT, execution of the last operation will not be resumed (see Appendix A). Note that any other command given during the DMA operation will also abort the operation in progress and should be avoided.

### 8.0 8044 IN EXPANDED OPERATION

To increase the number of information bytes in a frame, the 8044 can be operated in Expanded mode. In Expanded operation the system memory can be used as the transmit and receive buffer instead of the 8044 internal RAM. AP-283, "Flexibility in Frame Size Operation with the 8044", describes Expanded operation in detail.

### 8.1 Transmitting a Message in Expanded Operation

In Expanded operation the 8044 transmits the frame while it is fetching the data from the system memory using DMA. An internal transmit buffer is not necessary. The system memory can be used as the transmit buffer by the 8044.

Upon receiving the Transmit command, the 8044 enables the SIU and fetches the first data byte from the Command/Data register. The SIU transmits the opening flag, station address, and the control byte if the frame format includes these fields. It then transmits the fetched data. The 8044 CPU fetches the next byte while the previously fetched byte is being transmitted by the SIU. The CPU fetches the remaining bytes using DMA, then the SIU transmits them simultaneously until the end of message is reached. The SIU then transmits the FCS bytes, the closing flag and interrupts the 8044 CPU. The 8044 updates the status with the Transmit-Done event and interrupts the 80186. If the DMA does not keep up with transmission, the transmission is an underrun.

### 8.2 Receiving a Message in Expanded Operation

In Expanded operation the DMA controller transfers data to the system memory while the 8044 SIU is receiving them.

To receive a message, the 80186 allocates a block of memory for storing the message. It sets the DMA channel and sends the Receive command to the 8044.

Upon reception of the command, the 8044 enables its serial channel and waits for a frame. The SIU performs flag detection, address filtering, zero bit deletion, NRZI decoding, and CRC checking as it does in Normal operation.

After the SIU receives the first byte of the frame, the 8044 CPU requests the transfer of the byte to memory using DMA. The 80186 DMA moves the information byte into the system memory while the SIU is receiving the next byte. The next byte is transferred to the memory after the SIU receives it. When the entire frame is received, the SIU checks the received Frame Check Sequence bytes. If there is no CRC error, the SIU updates the 8044 registers and interrupts the 8044 CPU. The CPU updates the status and interrupts the 80186.

### 9.0 CONCLUSION

This application note describes an efficient way to interface the 80186 and the 80188 microprocessors to the Intel 8-bit microcontrollers like the 80C51, 80S52, and 8044. To illustrate this point the 80186 microprocessor interface to the 8044 microcontroller based serial communication chip was described. The hardware interface given here is very general and can interface the 8-bit microcontrollers to a variety of Intel microprocessors and DMA controllers. The microcontrollers with this interface hardware have the same benefits as both the Intel UPI-41/42 family and data communication peripheral chips such as the 82588 and the 82568 LAN controllers. Like the Intel UPI chips, they can be easily interfaced to microprocessors, and like the data communication peripherals, they execute high level commands. A similar approach can be used to interface Intel microprocessors to the 16-bit 8096 microcontroller.
APPENDIX A
SOFTWARE

The software modules shown here implement the execution of commands and status explained in sections 5 and 7. The 80186 software provides procedures to send commands and read status. The 8044 software decodes and executes the commands, updates the status, and interrupts the 80186. The procedures given here are called by higher level software drivers. For example, an 80186 application program may use the Transmit command to send a block of data to an application program that resides in the 8044 ROM or in another remote node. The application programs and the drivers that perform the communication tasks run asynchronously since all communication tasks are interrupt-driven.

Figure A-1 shows how to assign the ports and control registers for an 80186-based system. The software is written for an Intel iSBC® 186/51 computer board. The 8044 hardware is connected to the computer board iSBX™ connector.

Figure A-2 shows the 80186 command procedures. These procedures are used by the data link driver.

Figure A-3 shows how the DMA controller is loaded and initialized for data and parameter transfer from the 80186 memory to the 8044. This procedure is used by the TRANSMIT and CONFIGURE commands.

Figure A-4 shows how the DMA controller is loaded and initialized for data and parameter transfer from the 8044 to the 80186 memory. This procedure is used by the RECEIVE and DUMP commands.

Figure A-5 shows an interrupt service routine which handles interrupts resulting from various events. Note that this routine is not complete. The user should write the software to respond to events.

Figure A-6 shows an example of the 80186 software. It shows how to start various operations. This is not a data link driver, but it gives the procedures needed to write a complete driver.

Figure A-7 shows how to initialize the 8044. The user application program should be inserted here.

Figures A-8 through A-13 show the 8044 external interrupt service routine. In this routine a command received from the 80186 is decoded, and one of the command procedures shown in Figures A-9 through A-13 is executed.

Figure A-14 shows the serial channel (SIU) interrupt service routine. Note that execution of TRANSMIT, RECEIVE, and TRANSMIT-DISABLE commands are completed in this routine.
NAME COM_DRIVER
/** 80186 SOFTWARE FOR THE 80186/MICROCONTROLLER INTERFACE
 */ 8044 BOARD CONNECTED TO THE SBX1 OF THE SBC 186/51 BOARD.
/** SBX1 INTO TIE TO 80130 IR[0-7]. CONNECT JUMPER 30 TO 46.
 /** 80186 DMA CHANNEL 1 USED. CONNECT JUMPER 202 TO 203.

| TRUE  | EQU 0FFFFH | ADDRESS OF THE COMMAND REGISTER |
| FALSE | EQU 0H     | ADDRESS OF THE DATA REGISTER    |

; 8044 REGISTERS
CMD_44  EQU 080H
ST_44   EQU 080H
DATA_44 EQU 004H

; EVENTS
CON_DONE EQU 01H
TRA_DONE EQU 02H
DUM_DONE EQU 03H
REC_DONE EQU 04H
REC_DISA EQU 05H
TRA_DISA EQU 06H
ABO_DONE EQU 07H

; COMMANDS (INTA=1)
ABO_CMD  EQU 080H
REC_DIS_CMD EQU 081H
XMIT_DIS_CMD EQU 082H
REC_CMD   EQU 083H
TRA_CMD   EQU 084H
DUM_CMD   EQU 085H
CON_CMD   EQU 086H
NOP_CMD   EQU 087H

; 80186 DMA CHANNEL 1 REGISTERS
SL_DMA1  EQU 0FFDOH
SH_DMA1  EQU 0FFD2H
DL_DMA1  EQU 0FFD4H
DH_DMA1  EQU 0FFD6H
CTR_DMA1 EQU 0FFDAH

; 80186 INTERRUPT CONTROLLER REGISTERS
CTLO_INTR EQU 0FF38H
CTLI_INTR EQU 0FF3AH
MASK_INTR EQU 0FF28H
EOI_INTR  EQU 0FF22H
NSPEC_BIT EQU 08000H

; 80130 INTERRUPT CONTROLLER REGISTERS
EOI_DISR  EQU 060H
MASK_DISR  EQU 062H
RD_ISR  EQU 010H
RD_ISR  EQU 011H

IV_BASE  EQU 20H

Figure A-1. Port and Register Definitions for 80186 System
; INTERRUPT TABLE
INTERRUPTS SEGMENT AT 0
ORG (IV_BASE+1)*4H
IV_INTR O LABEL DWORD ; IRI VECTOR
INTERRUPTS ENDS

;******************************************************************************
STACK SEGMENT STACK 'STACK'
THE_STACK DW 200H DUP(?)
TOS LABEL WORD
STACK ENDS

;******************************************************************************
DATA SEGMENT PUBLIC 'DATA'
REC_BUFFER DB 1024 DUP(?)
CON_BUFFER DB 08H,01H,00H,00D0,55H,20H,05H,05H
DUM_BUFFER DB 0FH DUP(?)
TRA_BUFFER DB 07H,55H,11H,01H,02H,03H,04H,05H
CMND_FLAG DW FALSE
DATA ENDS

Figure A-1. Port and Register Definitions for 80186 System (Continued)

;******************************************************************************
CODE SEGMENT PUBLIC 'CODE'
ASSUME CS:CODE,
& DS:DATA,
& ES:NOTHING,
& SS:STACK

;******************************************************************************
RECV_COMMAND PROC FAR
PUSH BP
MOV BP,SP
LES SI,DWORD PTR [BP+6] ; LOAD BUFFER POINTER
MOV AX,WORD PTR[BP+10] ; LOAD BUFFER SIZE
MOV AH,0H
CALL REC_DMA ; CALL REC-INFO
MOV AL,REC_CMD ; LOAD RECEIVE COMMAND
OUT CMD_44,AL ; SEND TO COMMAND/DATA REG
POP BP
RET

RECV_COMMAND ENDP

;******************************************************************************
XMIT_COMMAND PROC FAR
PUSH BP
MOV BP,SP
LES SI,DWORD PTR [BP+6] ; LOAD BUFFER POINTER
MOV AX,WORD PTR[BP+10] ; LOAD BUFFER SIZE
MOV AH,0H
CALL TRA_DMA ; CALL TRA-INFO
MOV AL,TRA_CMD ; LOAD TRANSMIT COMMAND
OUT CMD_44,AL ; SEND TO COMMAND/DATA REG
POP BP
RET

XMIT_COMMAND ENDP

Figure A-2. Setup and Execution of Commands

231784-12

5-117
Figure A-2. Setup and Execution of Commands (Continued)
:*** RECEIVE DMA ***
: ARGX AX BUFFER SIZE
: ES:SI BUFFER POINTER

RECEIVE_DMA PROC NEAR
MOV DX,CNT_DMA1
; LOAD ADD OF TRANSFER COUNT REG
OUT DX,AX
; PROGRAM TRANSFER COUNT REGISTER
XOR BX,BX
; CLEAR BX
MOV AX,ES
; LOAD SEG ADDRESS OF BUFFER
SHL AX,1
RCL BX,1
SHL AX,1
RCL BX,1
SHL AX,1
RCL BX,1
ADD AX,SI
; ADD THE OFFSET TO BASE
ADC BX,0
MOV DX,DL_DMA1
; LOAD ADDRESS OF DEST POINTER (LO WORD)
OUT DX,AX
; PROGRAM DEST POINTER REGISTER (LO WORD)
MOV AX,BX
; LOAD ADDRESS OF SOURCE POINTER (HI WORD)
MOV DX,SH_DMA1
; LOAD ADDRESS OF SOURCE POINTER (LO WORD)
OUT DX,AX
; PROGRAM SOURCE POINTER REGISTER (HI WORD)
XOR AX,AX
; CLEAR AX
MOV AX,DATA_44
; LOAD ADDRESS OF DATA REGISTER
MOV DX,SL_DMA1
; LOAD ADDRESS OF SOURCE REGISTER
OUT DX,AX
; PROGRAM DEST POINTER REGISTER (HI WORD)
XOR AX,AX
; CLEAR AX
MOV DX,SH_DMA1
; LOAD ADDRESS OF CONTROL REGISTER
OUT DX,AX
; PROGRAM THE CONTROL REGISTER
MOV AX,101001010100110B
; LOAD THE CONTROL WORD
OUT DX,AX
; PROGRAM THE CONTROL REGISTER
RET

RECEIVE_DMA ENDP

Figure A-3. Loading and Starting the 80186 DMA Controller

:*** TRANSMIT DMA ***
: ARGX AX BUFFER SIZE
: ES:SI BUFFER POINTER

TRANSMIT_DMA PROC NEAR
INC AX
MOV DX,CNT_DMA1
; LOAD ADD OF TRANSFER COUNT REG
OUT DX,AX
; PROGRAM TRANSFER COUNT REGISTER
XOR BX,BX
; CLEAR BX
MOV AX,ES
; LOAD SEG ADDRESS OF BUFFER
SHL AX,1
RCL BX,1
SHL AX,1
RCL BX,1
SHL AX,1
RCL BX,1
ADD AX,SI
; ADD THE OFFSET TO BASE
ADC BX,0
MOV DX,SL_DMA1
; LOAD ADDRESS OF SOURCE POINTER (LO WORD)
OUT DX,AX
; PROGRAM SOURCE POINTER REGISTER (LO WORD)
MOV AX,BX
; LOAD ADDRESS OF SOURCE POINTER (HI WORD)
MOV DX,SH_DMA1
; LOAD ADDRESS OF SOURCE POINTER (HI WORD)
OUT DX,AX
; PROGRAM SOURCE POINTER REGISTER (HI WORD)
MOV AX,DATA_44
; LOAD ADDRESS OF DATA REGISTER
MOV DX,SL_DMA1
; LOAD ADDRESS OF SOURCE REGISTER
OUT DX,AX
; PROGRAM DEST POINTER REGISTER (LO WORD)
XOR AX,AX
; CLEAR AX
MOV AX,SH_DMA1
; LOAD ADDRESS OF CONTROL REGISTER
OUT DX,AX
; PROGRAM THE CONTROL REGISTER
MOV AX,101001010100110B
; LOAD THE CONTROL WORD
OUT DX,AX
; PROGRAM THE CONTROL REGISTER
RET

TRANSMIT_DMA ENDP

Figure A-4. Loading and Starting the 80186 DMA Controller
**AP-286**

---

### 80186 Interrupt Routine

**INT_186:**

1. **PUSH AX**
2. **PUSH DX**
3. **MOV AX,NSPEC_BIT** ; SEND NSPEC END OF INT
4. **MOV DX,EQI_INTR**
5. **OUT DX,AX**
6. **MOV AL,0110001B**
7. **OUT EOI_SINTR,AL**
8. **IN AL,ST_44** ; READ THE STATUS
9. **AND AX,OFFH**

; DECODE STATUS AND TAKE APPROPRIATE ACTION

10. **MOV DX,CTL_DMA1** ; DISABLE DMA
11. **IN AX,DX**
12. **OR AX,0100B**
13. **AND AX,NOT 010B**
14. **OUT DX,AX**
15. **MOV CMD_FLAG,TRUE**
16. **POP DX**
17. **POP AX**
18. **IRET**

---

**Figure A-5. Interrupt Service Routine**

---

;****************************************************

**BEGIN:**

1. **CLI**
2. **CLD**
3. ; SET ALL REGISTERS SMALL MODEL
4. **MOV SP,DATA**
5. **MOV DS,SP**
6. **MOV ES,SP**
7. **MOV SP,STACK**
8. **MOV SS,SP**
9. **MOV SP,OFFSET TOS**

; SETUP INTERRUPT VECTORS

10. **PUSH ES**
11. **XOR AX,AX**
12. **MOV ES,AX**
13. **MOV WORD PTR ES:IV_INTR +1, OFFSET INT_186**
14. **MOV WORD PTR ES:IV_INTR +2, CS**
15. **POP ES**

; SETUP 80130 INTERRUPT CONTROLLER

16. **MOV AL,00010011B** ; ICW1
17. **OUT EOI_SINTR,AL**
18. **MUL AL**
19. **MOV AL,IV_BASE** ; ICW2
20. **OUT MASK_SINTR,AL**
21. **MUL AL**
22. **MOV AL,00000000B** ; ICW4
23. **OUT MASK_SINTR,AL**
24. **MUL AL**
25. **MOV AL,OFCH** ; MASK
26. **OUT MASK_SINTR,AL**

---

**Figure A-6. Example of Executing Commands**

---

5-120
Figure A-6. Example of Executing Commands (Continued)
$DEBUG NOMODS1
$INCLUDE (REG44.PDF)

; THE 8044 SOFTWARE DRIVER FOR THE 80186/8044 INTERFACE.
ORG 00H
SJMP INIT
JMP EINT0
ORG 23H
JMP INT

;*************** INITIALIZATION ***************

ORG 26H
INIT:
MOV TCON,#00000001B ; EXT INTO: EDGE TRIGGER
MOV IE,#00010001B ; SI=EXO=1
CLR P1.1 ; CLEAR IRQ STATUS BIT
SETB EA ; ENABLE INTERRUPTS
DOT: SJMP DOT ; WAIT FOR AN INTERRUPT

Figure A-7. Initialization Routine

;***************EXTERNAL INTERRUPT 0 ***************
EINT0: CLR P1.5 ; CLEAR THE E BIT
MOV DPTR,#100H ; LOAD DATA POINTER WITH A DUMMY NUMBER
MOVX A,#00H ; READ THE COMMAND BYTE.
ANL A,#00011111B ; KEEP THE OPERATION FIELD
MOV R2,A ; SAVE COMMAND

; DECODE COMMAND AND JUMP TO THE APPROPRIATE ROUTINE
; COMMAND OPERATION (BIT5-3)
;
J1: CJNE A,#00H,J2
JMP CABO
J2: CJNE A,#01H,J3
JMP CABO
J3: CJNE A,#02H,J4
JMP CRCIS
J4: CJNE A,#03H,J5
JMP CTDIS
J5: CJNE A,#04H,J6
JMP CTBA
J6: CJNE A,#05H,J7
JMP CUMP
J7: CJNE A,#06H,J8
JMP CCON
J8: CJNE A,#07H,J9
JMP CHOP
J9: RETI

; IF INTO IS SET TO PRIORITY 1,
; THEN DMA OPERATION WAS IN PROGRESS.
; EXECUTE ABORT REGARDLESS OF THE
; COMMAND ISSUED.
J1: CJNE A,#00H,J2
JMP CABO

; THIS LINE WILL BE EXECUTED IF ABORT WAS
; ISSUED WHEN THE 8044 IS NOT EXECUTING
; ANY COMMANDS.
J1: CJNE A,#01H,J3
JMP CRCIS
J3: CJNE A,#02H,J4
JMP CTDIS
J4: CJNE A,#03H,J5
JMP CRCIS
J5: CJNE A,#04H,J6
JMP CTBA
J6: CJNE A,#05H,J7
JMP CUMP
J7: CJNE A,#06H,J8
JMP CCON
J8: CJNE A,#07H,J9
JMP CHOP
J9: RETI

Figure A-8. External Interrupt Service Routine
; ** NOP COMMAND
CNOP: CLR IE0 ; IGNORE PENDING EXT INTO (IF ANY).
; ANY INTERRUPT (COMMAND) DURING
; EXECUTION OF AN OPERATION IS IGNORED
RETI ; RETURN

; ** ABORT COMMAND
CABO: JNB PXO,CABOJ1 ; WAS DNA IN PROGRESS?
CLR PXO ; YES. EXT INTO: PRIORITY 0
CLR Pl.1 ; CLEAR DMA REQUEST
SETB P1.2 ; UPDATE STATUS WITH
SETB P1.3 ;ABORT-DONE EVENT
SETB P1.4 ; (STATUS=D8H: E=0)
CLR IE0 ; IGNORE PENDING EXT INTO (IF ANY).
CLR P1.0
SETB P1.0 ; SET INT BIT AND INTERRUPT 80186
JB P3.2, $ ; WAIT TILL INTERRUPT IS ACKNOWLEDGED
; EXECUTE THE NEXT "RETI" TWICE
POP ACC ; POP OUT THE OLD HI BYTE PC
POP ACC ; POP OUT THE OLD LOW BYTE PC
MOV B, $HIGH($+10) ; HI BYTE ADDRESS OF CABOJ2
MOV ACC, $LOW($+7) ; LOW BYTE ADDRESS OF CABOJ2
; PUSH THE ADDRESS OF THE NEXT
PUSH B ; "RETI" INSTRUCTION INTO STACK
CABOJ2: RETI ; RETURN

CABOJ1: NOP ; DMA WAS NOT IN PROGRESS
SETB P1.5 ; SET THE E BIT
SETB P1.2 ; UPDATE STATUS WITH
SETB P1.3 ;ABORT-DONE EVENT
SETB P1.4 ; (STATUS=F8H: E=1)
CLR IE0 ; IGNORE PENDING EXT INTO (IF ANY).
CLR P1.0
SETB P1.0 ; SET INT BIT AND INTERRUPT 80186
JB P3.2, $ ; WAIT TILL INTERRUPT IS ACKNOWLEDGED
RETI ; RETURN

Figure A-9. Execution of NOP and ABORT Commands

; ** CONFIGURE COMMAND
CCON: MOV DPTR, #100H ; IGNORE PENDING EXT INTO (IF ANY)
CLR IE0 ; EXT INTO: PRIORITY 1
SETB PXO ; PXO IS SET TO ACCEPT ABORT
; DURING DMA OPERATION.
; ENABLE DMA REQUEST
SETB P1.1
JB P3.3, $ ; WAIT FOR DMA ACK.
MOVX A, @DPTR ; READ FROM COMMAND/DATA REGISTER
MOV R0, A ; LOAD BYTE COUNT
DEC R0 ; DECREMENT BYTE COUNT
JS P3.3, $ ; WAIT FOR DMA ACK.
MOVX A, @DPTR ; READ FROM COMMAND/DATA REGISTER
MOV R1, A ; LOAD FIRST-BYTE
JS P3.3, $ ; WAIT FOR DMA ACK.
MOVX A, @DPTR ; READ FROM COMMAND/DATA REGISTER
CJNE R1, #01H, CCONF1 ; CHECK THE FIRST-BYTE
MOV SMD, A ; UPDATE THE SMD REGISTER
INC R1 ; INC. POINTER TO THE CONF. BLOCK
INC R0, CCONF4 ; CHECK THE BYTE COUNT
JMP CCON1

CCONF4: JB P3.3, CCONF4
MOVX A, @DPTR
CCONF1: CJNE R1, #02H, CCONJ2
MOV SMD, A
INC R1
DJNZ R0, CCONF5
JMP CCON1

CCONF5: JB P3.3, CCONF5
MOVX A, @DPTR
CCONJ2: CJNE R1, #03H, CCONJ3
MOV SMD, A
INC R1
DJNZ R0, CCONF6
JMP CCON1

CCONF6: JB P3.3, CCONF6
MOVX A, @DPTR
CCONJ3: CJNE R1, #04H, CCONJ4

Figure A-10. Execution of CONFIGURE Command
MOV TRS, A
INC R1
DJNZ R0, CCONF7
JMP CCONFI

CCONF7: JB P3.3, CCONF7
MOVX A, @DPTR

CCONF4: CJNE R1, $00H, CCONF5
MOV TBL, A
INC R1
DJNZ R0, CCONF6
JMP CCONFI

CCONF8: JB P3.3, CCONF8
MOVX A, @DPTR

CCONF5: CJNE R1, $00H, CCONF6
MOV RBS, A
INC R1
DJNZ R0, CCONF9
JMP CCONFI

CCONF9: JB P3.3, CCONF9
MOVX A, @DPTR

CCONF6: CJNE R1, $00H, CCONF7
MOV RBL, A
INC R1
DJNZ R0, CCONF10
JMP CCONFI

CCONF A: JB P3.3, CCONF A
MOVX A, @DPTR

CCONF7: CJNE R1, $00H, CCONF8
MOV IP, A
INC R1
DJNZ R0, CCONF8
JMP CCONFI

CCONF B: JB P3.3, CCONF B
MOVX A, @DPTR

CCONF8: CJNE R1, $00H, CCONF9
MOV IE, A
INC R1
DJNZ R0, CCONF11
JMP CCONFI

CCONF C: JB P3.3, CCONF C
MOVX A, @DPTR

CCONF9: CJNE R1, $00H, CCONF10
MOV TMOD, A
INC R1
DJNZ R0, CCONF10
JMP CCONFI

CCONF D: JB P3.3, CCONF D
MOVX A, @DPTR

CCONF A: CJNE R1, $00H, CCONF B
MOV TCON, A
INC R1
DJNZ R0, CCONF B
JMP CCONFI

CCONF E: JB P3.3, CCONF E
MOVX A, @DPTR

CCONF B: CJNE R1, $00H, ERROR1
MOV PSW, A
INC R1
DJNZ R0, ERROR1
JMP CCONFI

ERROR1: NOP
SETB P1.5 ; ILLEGAL BYTE COUNT
; SET THE E STATUS BIT

CCONFI: NOP
CLR P1.1 ; CLEAR DMA REQUEST
CLR PX0 ; EXT INTO: PRIORITY 0

SETB P1.2 ; UPDATE STATUS WITH
CLR P1.3 ; CONFIGURE-DONE EVENT
CLR P1.4 ; (STATUS=CSH IF E=0)
CLR IE0 ; IGNORE PENDING EXT INTO (IF ANY)
CLR P1.0
SETB P1.0 ; INTERRUPT THE 80186
JB P3.2, $ ; WAIT TILL INTERRUPT IS ACKNOWLED
RETI

Figure A-10. Execution of CONFIGURE Command (Continued)
; ** DUMP COMMAND

CDUMP:  MOV A,STS
        MOVX @DPTR,A
        CLR IE0
        SETB PX0
        SETB P1.1
        JB P1.3,$
        MOV A,SMO
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,STAD
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TBL
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,SMO
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,STAD
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TBL
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,SMO
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,STAD
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TBL
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,SMO
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,STAD
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TBL
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,SMO
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,STAD
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TBL
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,SMO
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,STAD
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TBL
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,SMO
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,STAD
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TBL
        MOVX @DPTR,A
        JB P1.3,$
        MOV A,TDB
        MOVX @DPTR,A
        JB P1.3,$
        CLR P1.1
        CLR PX0
        SETS P1.2
        SETS P1.3
        CLR P1.4
        CLR IE0
        CLR P1.0
        SETS P1.0
        JB P1.2,$
        RETI
        RETI

Figure A-11. Execution of DUMP Command
; ** RECEIVE COMMAND.
CREC: JNB RBE,CRECJ1
SETB P1.5
; YES. SET THE E BIT
CRECJ1: SETB RBE
; NO. ENABLE RECEPTION
CLR RBP
; CLEAR RECEIVE BUFFER PROTECT BIT
CLR IE0
; RETURN. UPDATE STATUS IN THE SIU INTERRUPT ROUTINE.

; ** TRANSMIT COMMAND.
CTRA: MOV R1,TBS
; LOAD TRANSMIT BUFFER START
CLR IE0
; IONE PENDING EXT INTU (IF ANY)
SETB PX0
; EXT INTO: PRIORITY 1
SETB P1.1
; ENABLE DMA REQUEST
JB P3.2,6
; WAIT FOR DMA ACK.
MOVX A, @DPTR
; READ FROM COMMAND/DATA REG.
MOV RO, A
; LOAD THE BYTE COUNT
DEC A
; DECREMENT THE BYTE COUNT
DEC D
; COUNT AND LOAD INTO XMIT
MOV TBL, A
; LOAD BUFFER LENGTH
CTRAJ2: JB P3.3,CTRAJ2
MOVX A, @DPTR
; READ FROM COMMAND/DATA REG.
MOV STAD, A
; LOAD DESTINATION ADDRESS
DEC RO
; DECREMENT THE BYTE COUNT
CTRAJ3: JB P3.3,CTRAJ3
MOVX A, @DPTR
; READ FROM COMMAND/DATA REG.
MOV TCB, A
; LOAD THE TRANSMIT CONTROL BYTE
DJNZ RO, CTRAJ4
; IS THERE ANY INFO. BYTE?
SJMP CTRAJ5
; NO.
CTRAJ4: JB P3.3,CTRAJ4
; YES. WAIT FOR DMA ACK.
MOVX A, @DPTR
; READ FROM COMMAND/DATA REG.
MOV RX0, A
; MOVE DATA TO THE TRANSMIT BUFFER
INC R1
; INC. POINTER TO BUFFER
DJNZ RO, CTRAJ4
; LAST BYTE FETCHED INTO THE BUFFER?
; NO. FETCH THE NEXT BYTE
CTRAJ5: CLR P1.1
; YES. DISABLE DMA REQUEST
CLR PX0
; EXT INTO: PRIORITY 0
SETB TBF
; SET TRANSMIT BUFFER FULL
SETB RRS
; ENABLE TRANSMISSION
CLR IE0
; RETURN. UPDATE STATUS WITH RECEIPTION-DISABLED EVENT
RETI
; SIU INTERRUPT ROUTINE

; ** TRANSMIT-DISCONNECT COMMAND
CTDIS: JNB TBF,CTDISJ1
; IS TRANSMIT BUFFER ALREADY EMPTY?
SETB P1.5
; YES. SET THE E BIT
CTDISJ1: CLR TBF
; NO. CLEAR TRANSMIT BUFFER
CLR IE0
; RETURN. UPDATE STATUS IN THE SIU INTERRUPT ROUTINE.

; ** RECEIVE-DISCONNECT COMMAND
CRDIS: JNB RBE,CRDISJ1
; IS RECEIVE BUFFER ALREADY EMPTY?
SETB P1.5
; YES. SET THE E BIT
CRDISJ1: CLR RBE
; NO. CLEAR RECEIVE BUFFER
SETB P1.2
; UPDATE STATUS WITH RECEPTION-DISABLED EVENT
CLR P1.3
; (STATUS=D5 IF E=0)
CLR IE0
CLR P1.0
SETB P1.0
; INTERRUPT THE 80186
JB P3.2,6
; WAIT UNTIL INTERRUPT IS ACKNOWLEDGED
RETI
; RETURN.

Figure A-12. Execution of RECEIVE and TRANSMIT Commands

Figure A-13. Execution of RECEIVE-DISCONNECT and TRANSMIT-DISCONNECT Commands
;*************** SERIAL CHANNEL (SIU) INTERRUPT ***************

SIINT:  CLR SI
MOV A,R2 ; LOAD THE OPERATION FIELD
CJNE A,'O3H,SINTJ1 ; RECEIVE COMMAND PENDING?
JMP SIREC ; YES.
SINTJ1: CJNE A,'O2H,SINTJ2 ; TRANSMIT-DISCONNECT PENDING?
JMP SITDIS ; YES.
SINTJ2: JMP SITRA ; TRANSMIT COMMAND IS PENDING

;** TRANSMISSION IS DISABLED

SITDIS: JB RTS,SINTJ3 ; REQUEST TO SEND ENABLED?
JNB TBF,SINTJ4 ; YES. TRANSMISSION DISABLED?
CLR P1.2 ; YES. UPDATE STATUS WITH
SETB P1.3 ; TRANSMISSION-DISABLED EVENT
SETB P1.4 ; (STATUS=D9H)
CLR IE0 ; IGNORE PENDING EXT INTO
CLR P1.0 ; INTERRUPT THE 80186
JB P1.2,$ ; WAIT TILL INTERRUPT IS ACKNOWLEDGED
RETI

;** A FRAME IS TRANSMITTED

SITRA: JB RTS,SINTJ3 ; A FRAME TRANSMITTED?
CLR P1.2 ; YES.
CLR IE0 ; UPDATE STATUS WITH
SETB P1.3 ; TRANSMIT-DONE EVENT
CLR P1.4 ; (STATUS=C9)
CLR P1.0 ; INTERRUPT THE 80186
JB P1.2,$ ; WAIT TILL INTERRUPT IS ACKNOWLEDGED
RETI

;** A FRAME IS RECEIVED

SIREC: JB RBE,SINTJ3 ; RECEIVE BUFFER FULL?
JNB BOV,SINTJ4 ; YES. BUFFER OVERFLOW?
SETB P1.5 ; YES. SET THE B BIT
SINTJ4: MOV R0,RFL ; LOAD R0 WITH RECEIVE BYTE COUNT
MOV R1,RBS ; LOAD R1 WITH RECEIVE BUFFER ADDRESS
CLR IE0 ; IGNORE PENDING EXT INTO (IF ANY)
SETB PX0 ; EXIT INTO: PRIORITY 1
MOV A,R01 ; MOVE FIRST BYTE INTO ACC.
MOVC @DPTR,A ; WRITE TO THE COMMAND/DATA REG
SETB P1.1 ; ENABLE DMA REQUEST
INC R1 ; INC POINTER TO RECEIVE BUFFER
JB P3.3,$ ; WAIT FOR DMA ACK.
DJNZ RO,CINTJ7 ; LAST BYTE MOVED?
SJMP CINTJ8 ; YES

CINTJ7: MOV A,R01 ; LOAD RECEIVED DATA INTO ACC.
MOVC @DPTR,A ; WRITE TO THE COMMAND/DATA REG
INC R1 ; INC POINTER TO RECEIVE BUFFER
JB P3.3,$ ; WAIT TILL DMA ACK
DJNZ R0,CINTJ7 ; LAST BYTE MOVED TO COMMAND/DATA REG?
SJMP CINTJ8 ; NO. DEPOSIT THE NEXT BYTE

CINTJ8: MOV A,RFL ; LOAD BYTE COUNT
MOVC @DPTR,A ; WRITE TO THE COMMAND/DATA REG
JB P3.3,$ ; WAIT FOR DMA ACK.
MOV A,STAD ; LOAD STATION ADDRESS
MOVC @DPTR,A ; WRITE TO THE COMMAND/DATA REG
JB P3.3,$ ; WAIT FOR DMA ACK.
MOV A,RBC ; LOAD RECEIVE CONTROL BYTE
MOVC @DPTR,A ; WRITE TO THE COMMAND/DATA REG
JB P3.3,$ ; WAIT FOR DMA ACK.
CLR P1.1 ; CLEAR DMA REQUEST
CLR PX0 ; EXTERNAL INTERRUPT: PRIORITY 0

Figure A-14. Serial Channel Interrupt Routine
CLR P1.2 ; UPDATE STATUS WITH
CLR P1.3 ; RECEIVE-DONE EVENT
SETB P1.4 ; (STATUS=O1H IF E=0)
CLR IE0 ; IGNORE PENDING EXT INTO
CLR P1.0 ; INTERRUPT THE 80186
JB P1.2,$ ; WAIT TILL INTERRUPT IS ACKNOWLEDGED
RETI

SINTJ3: NOP
RETI

END

Figure A-14. Serial Channel Interrupt Routine (Continued)
80186/80188 DMA Latency

STEVE FARRER
APPLICATIONS ENGINEER
When using the DMA controller of the 80186 and 80188, there are several operating conditions which affect the service time (latency) between when the DMA request is generated and when the bus cycles associated to the DMA transfer are actually run. This application brief describes those conditions which affect DMA Latency.

**DMA REQUEST GENERATION**

The minimum DMA latency is 4 clocks and, depending on when the signal arrives (i.e. if the signal just missed the setup time), it might appear to be almost 5 clocks. This 4 to 5 clock delay is due to a two phase synchronizer and various transfer gate delays the DRQ signal must take before reaching the BIU. Conceptually the circuit looks like Figure 1.

If the Bus Interface Unit (BIU) is available when the DRQ signal reaches it, then a DMA cycle will proceed at T1 of the bus cycle as the next clock.

Also note that the DRQ signal is not latched, and must remain active until serviced. If the DRQ signal is brought low after being asserted high, then a '0' will propagate through and; if the request had not yet been serviced, then the BIU will see a '0' and the cycle will never take place.

**Conditions Affecting DMA Latency**

The circumstances that affect DMA latency in order of worst case are as follows:

1) HOLD
2) LOCK - INTA
3) Odd byte accesses
4) Effective Address Calculations (EA)

HOLD can indefinitely delay a DMA cycle. There is no mechanism internally to remove HLDA when a DMA request is pending.

LOCKed instructions can also delay a DMA cycle by a significant amount, depending on the type of instruction locked. A typical locked XCHG instruction from memory to register could delay the DMA cycle by as much as 18 clocks if the memory access required two bus cycles (80188 or odd locations on the 80186). On the other hand, a locked repeat MOVS could delay a DMA cycle by up to 1.05 million clocks depending on the number of transfers and the number of bus cycles per transfers.

Interrupt acknowledges can also affect DMA latency because the bus is locked out during the first two bus cycles required to fetch the interrupt vector type. This causes the worst case latency during interrupt acknowledges to be:

<table>
<thead>
<tr>
<th>Worst Case Latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Clocks (Minimum Setup)</td>
<td>Holds at T1</td>
</tr>
<tr>
<td>10 Clocks (2 Bus Cycles + 2 Idle Clocks)</td>
<td>Holds at T1 + 2 idle</td>
</tr>
<tr>
<td>14 Clocks Total</td>
<td>Holds at T1 + 2 idle</td>
</tr>
</tbody>
</table>

Both HOLD and LOCK are extremely dependent on the type of system being designed and therefore are not really considered to be normal worst case latency. However, odd byte accesses and effective address calculations are conditions that frequently occur in almost all systems. Under these conditions of no HOLD, no LOCK, and no wait states, the worst case occurs when the DMA request loses to an instruction data cycle requiring an effective address calculation.

Effective addresses (EA) always require 4 clocks for calculation and can only take place during T3-T4-TI-TI, T4-TI-TI-TI, or T1-TI-TI-TI. This creates an extra minimum insertion of 2 T-idle cycles. If the EA requires an immediate value in the prefetch queue, then a signal goes active which places the EA bus cycle at a higher priority than any other BIU requests. This is so the execution unit won't be waiting on the bus interface unit. If the EA hadn't required the value in the queue, then the EU could proceed with the next instruction shortly after it had sent the request to the BIU. Figure 2 shows the effects EA calculations have on DMA Latency.

![Figure 1. DMA Request Synchronization](image-url)
Figure 2. Logic State Analyzer Trace and Accompanying Program Code

* If an immediate value had not been used, the EA would have been aborted and the DMA would have begun its bus cycle. In this case, the latency would be 8 clocks.
80186/80188 EFI Drive and Oscillator Operation

STEVE FARRER
APPLICATIONS ENGINEER

© Intel Corporation, 1987
Order Number: 270526-001
There has been some confusion in the past regarding the correct input for EFI (External Frequency Input) use and what parameters should be used for crystal selection. This Application Brief discusses the trade-offs with each input so that one can decide which input suits his design and also lists the parameters for crystal selection.

**EFI Operation**

The oscillator circuit on the 186/188 is as shown in Figure 1 (simplified). Either input may be used for an EFI signal. Using X1 requires very little drive from an external oscillator since it is essentially the gate of an NMOS transistor. Clock operation works fine using this input, but at higher frequencies the stray capacitance on X2 begins to change the duty cycle of the clock. This will eventually cause the part to fail.

**NOTE:** Driving X2 does not allow compatibility with future CMOS designs.

Using X2 as an EFI gives a broader frequency range but places a more stringent requirement on the drive capability of the external oscillator. Since X1 is an input, it may be grounded to minimize the capacitance. This in turn allows for a higher frequency range since the duty cycle remains closer to 50%. But with X1 grounded, the output of the inverter (which is directly connected to X2) is always trying to output a high. This means the oscillator driving X2 must be capable of sinking up to 15 mA at cold temperatures when trying to drive it low. If the external oscillator is capable of supplying 15 mA, then this method is preferred. Otherwise, X1 should be used as an EFI.

Caution: using X2 for EFI does not allow for CMOS compatibility at a future date.

**Crystal Operation**

The oscillator circuit is a single stage amplifier connected as a Pierce oscillator. There are no passive components in the oscillator circuit, only a unique combination of depletion and enhancement mode FET's. Characterization of the oscillator circuit showed that operation was optimum with crystal parameters as follows:

- **ESR** (Equivalent Series Resistance) 30 ohms maximum
- **C0** (Shunt Capacitance) 7.0 pf max.
- **C1** (Load Capacity) 20 pf ± 2 pf
- **Drive Level** 1 mW max.

This characterization data was supplied by:

<table>
<thead>
<tr>
<th>Standard Crystal Corporation</th>
<th>9940 East Baldwin Place</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>El Monte, CA 91731</td>
</tr>
<tr>
<td></td>
<td>(213) 443-2121</td>
</tr>
</tbody>
</table>

---

**Figure 1. Oscillator Circuit on the 186/188**

Using X2 as an EFI gives a broader frequency range but places a more stringent requirement on the drive.
The 80C186/80C188 Integrated Refresh Control Unit

GARRY MION
ECO SENIOR APPLICATIONS ENGINEER
The 80C186 and 80C188 incorporate a special control unit that integrates address and clock counters which, along with the Bus Interface Unit (BIU), facilitates dynamic memory refreshing. Refreshing is an operation required by dynamic memory to ensure data retention.

Dynamic memory refreshing can be controlled using anything from an exotic memory controller to a simple timer along with a DMA controller. In fact, the 80C186 device accomplishes the task memory refreshing by using one of the internal timer/counters and a DMA channel. However, doing this meant that very desirable internal functions were no longer available to do more useful work.

Dynamic memory, unlike static or non-volatile memory, always require some form of a memory controller to enable read and write operations. Therefore, even the most basic dynamic memory interface has a minimum set of support logic. The advent of programmable logic and highly integrated dynamic memory has made the job of designing a memory controller somewhat straightforward. However, directly supporting memory refresh still can complicate many controller designs.

The designer of a memory controller must take into account CPU-versus-refresh arbitration and must provide a mechanism to generate periodic refresh requests. Most dynamic memory devices now contain internal refresh address counters which eliminate the need for external refresh address generation. However, such devices tend to complicate a memory controller design. The 80C186 simplifies dynamic memory controller design by integrating a refresh mechanism into the operation of the CPU.

This application brief is not intended to be a discussion of dynamic memory controller design. Instead, it will concentrate on the operation of the Refresh Control Unit with the 80C186, and how it can help simplify a memory controller.

The discussions on the following pages apply to BOTH the 80C186 and 80C188 except where noted.

**UNDERSTANDING DYNAMIC MEMORY**

Before explaining how memory refreshing is accomplished, some understanding of a Dynamic Random Access Memory (DRAM) device is needed. Figure 1 shows a simplified block diagram of a DRAM device, while a block diagram of a typical dynamic memory controller is shown in Figure 2.

![Figure 1. Random Access Memory Device](image-url)
The typical DRAM memory array is built as a matrix. Thus, any bit or cell in the memory array is accessed by specifying a unique row and column address. As shown in Figure 1, the row and column addresses are multiplexed through one set of address inputs. Multiplexing the address inputs helps reduce the number of pins required to support large memory arrays. For instance, adding only one address bit will result in a memory array 4 times as large.

Two control lines, RAS and CAS, are used to strobe an address into the memory chip. Figure 3 illustrates a timing diagram for a typical memory read access and the relationship between the RAS and CAS signals. The signal MUX controls which half of the address is presented to the memory devices. After generating the row address strobe (RAS), the decoder selects a row of memory cells whose data value will be detected by a Sense Amplifier. The Sense Amplifier then presents the data to the column decoder. Note that all cells associated to a row get accessed. The fact that all cells within a row are accessed will be used later to explain why only the RAS portion of the memory address is required to refresh a device.
When the column address is strobed, it is here that only one of the memory cells is selected. The memory cell will either be written to or read from depending on the control signals WE and OE respectively. Since data from one entire row is presented to the column decoder, it is possible (on some devices) to simply cycle through column addresses to access additional data. The basic idea, however, is that two sets of addresses are required to access a memory cell within a memory array. Furthermore, specifying a single row address internally accesses all memory cells within that row.

The minimum memory controller interface consists of a sequencer and an address multiplexer. The sequencer is responsible for generating the correct control signals: RAS; CAS; MUX; WE; OE. The address multiplexer logic is responsible for translating the processor address bus to the memory address bus. These two pieces of logic can exist in any form, from simple TTL gates to single chip solutions. However, what is missing from the simplified memory controller is a mechanism to perform memory refresh.

UNDERSTANDING MEMORY REFRESH

As indicated earlier, dynamic memory needs to be refreshed in order to maintain its data. Refreshing is accomplished whenever a memory cell is accessed. It is not necessary to read a memory location and then write the value back in order to refresh a memory cell. Simply cycling through a complete set of row addresses is all that is required. Remember, since a row accesses all memory cells associated to it, accessing all rows will access all the cells within the device.

Referring back to Figure 2, the 9 address bits presented to the memory devices are multiplexed from the 18 bits of address generated by the 80C186. In the design, address bits A1-A9 are presented during RAS, while address bits A10-A18 are presented during CAS. Note that address bit A0 is not used because the memory array is organized as word wide; A0 along with BHE are used to select one or both of the bytes within a word.

Cycling through row addresses is the only requirement needed to refresh a DRAM device. Using the example in Figure 2, 9 bits of address are needed. Nine bits represent 512 unique addresses, and the only requirement is that each unique address be regenerated every 8 ms (maximum refresh rate for most devices with 512 rows). An 8 ms refresh interval divided by 512 addresses results in an average refresh cycle rate of 15.625 microseconds. Therefore, every 15.625 microseconds a mechanism must exist that will access the DRAM device, each time presenting a new row address. Any rate faster than 15.625 microseconds is acceptable, but significantly faster times have the potential of decreasing memory performance.

WAYS TO REFRESH A MEMORY DEVICE

For most dynamic memory devices, there are several ways in which a refresh cycle can be run. The first and simplest way is to generate memory read cycles every 15.6 microseconds. Each new memory read cycle would generate a unique address. When refreshing is accomplished using memory read cycles, the memory controller is simplified. Only the basic control signals need to be generated, which are the minimum needed to access the memory anyway. Simplicity is, however, accompanied by one drawback; bus overhead. Using memory reads to perform DRAM refreshing means that one bus cycle every 15.6 microseconds is wasted. When operating at very slow speeds, a wasted bus cycle might appear to be significant. But if a bus cycle takes only, say, 320 nanoseconds to complete, running a refresh cycle every 15.6 microseconds represents a two percent hit in bus performance.

A second method relies on the fact that most dynamic memory devices now have built-in refresh address mechanisms. DRAM refreshing can be accomplished by generating CAS before RAS signaling (see Figure 4a). This method requires that an external signal generate a periodic request to the DRAM controller to initiate the refresh cycle. A method similar to CAS before RAS refreshing is hidden refresh. Figure 4c illustrates the timing involved to perform hidden refresh. No request logic is needed, since the memory access itself is what initiates the refresh cycle. However, constant memory accessing is required in order to maintain refreshing. Once accessing stops, refreshing stops. Both of the methods described have the advantage of not consuming bus bandwidth, but require the memory controller to handle the somewhat different (from normal memory accessing) signaling requirements.
A final method is to implement a discrete design that supports refresh control and refresh address generation. The circuit details are shown in Figure 4b. A discrete design allows the most design flexibility and can be tailored to meet any system-to-memory interfacing requirements.

There are other methods available, most of which involve single-chip dedicated memory controllers. However, any memory controller design that performs the function of refreshing either directly or through external support circuitry has one major concern; arbitration between the refresh cycle and a normal memory access. The best way to make the operation of the DRAM memory controller a true slave to the operation of the CPU is to include refreshing as part of the functionality of the CPU. By offloading the task of memory refreshing onto the CPU, the memory controller can be simplified and dedicated to the duty of DRAM interfacing.

The idea that the 80C186 refresh cycle is simply a memory read means that the dynamic memory control logic does not need to differentiate between refresh cycles and normal memory read cycles. This simplifies the design of the memory controller. There are no special signaling requirements needed, and RAS only refreshing (for low-power designs) can be easily accommodated. Further, since the request is generated internally and synchronous with the operation of the BIU, no special external logic needs to detect when a refresh cycle conflicts with a CPU access.
80C186 REFRESH CONTROL FEATURES

The Refresh Control Unit (RCU) of the 80C186 consists of a 9-bit address counter, a 9-bit down counter, and support logic. The block diagram can be seen in Figure 5.

The 9-bit address counter is controlled by the BIU and used whenever a refresh bus cycle is executed. Thus, any dynamic memory device whose refresh address requirement does not exceed nine bits can be directly supported by the 80C186. A special register has been defined to allow the base (starting) address of the refresh memory region to be specified. This base address can be located on any 4 kilobyte boundary. Furthermore, if this refresh base address overlaps any of the defined chip select regions, the chip select defined for that region will go active.

The 9-bit down counter initiates a refresh request. When the counter decrements to 1 (it decrements every clock cycle), a refresh request is presented to the BIU. When the bus is free, the BIU will run the refresh (memory) bus cycle. Note that since a refresh bus cycle is executed by the BIU, the faster refresh cycles are requested the greater the impact on bus performance.

Referring back to the discussion of request rates, the maximum refresh period is typically 15.6 microseconds. With the 80C186 operating at 12.5 MHz, this represents a refresh bus impact of only 2%. However, at 5 microseconds the bus impact is 15%. Therefore, the refresh request rate should be tailored to meet the needs of the dynamic memory and the system. The 80C186 provides flexibility by allowing the request rate to be programmable in 80 ns steps (at 12.5 MHz).

To facilitate low power designs, the refresh bus cycle provides a mechanism whereby the dynamic memory devices can be turned off during refresh accesses. Low power control is accomplished by driving both address bit A0 and the control signal BHE to a high level. Essentially an invalid bus access condition exists, since A0 and BHE are used to indicate which half of the data bus is being accessed. When both are high during the access, the indication is that neither half of the bus is being used for the data transfer. This is acceptable for refresh bus cycles since no data is actually being transferred. If the memory controller takes advantage of this condition, the output enables of the dynamic memory devices (as well as the CAS strobe) can be disabled during refresh bus cycles, providing overall lower power consumption.

PROGRAMMING CHARACTERISTICS OF THE REFRESH CONTROL UNIT

A block of control registers are defined in the Peripheral Control Block (PCB) that define the operating characteristics of the refresh control unit (refer to Figure 5). These registers are only accessible when the 80C186 is operating in enhanced mode. When in compatibility mode, the 80C186 will ignore any reads or writes to the RCU registers.

The three registers associated with the refresh unit (MDRAM, CDRAM, EDRAM) provide the following features:
1) Enable/disable refresh unit
2) Establish a refresh request rate
3) Establish a refresh memory region
4) Examine the refresh down counter

It is not necessary to program any of these registers in a specific sequence, although the refresh request rate and refresh base address registers should be programmed before the refresh unit is enabled.
Programming the Memory Partition Register

The MDRAM register (Figure 6) is used to define address bits A13 through A19 of the 20 bit refresh address. This essentially establishes a memory region which will be accessed during refresh bus cycles. Typically, the refresh memory region will overlap a chip select that is used to access the dynamic memory. Overlapping the refresh memory region with a chip select memory region, means no additional external hardware is needed to support refresh bus cycles since it essentially operates the same as memory read cycles. When the 80C186 is reset, the MDRAM register is initialized to zero.

Figure 7 illustrates how the refresh address is generated. Address bits A10-A12 are not programmable and are always driven to a zero during a refresh bus cycle. Address bits A1 through A9 are derived by a 9-bit linear-feedback shift counter. The address counter is not ascending or contiguous, meaning that the counter does not start at 0 and increment to 511 before resetting back to 0. For refreshing purposes, it is not important that the address be contiguous and count up or down. Rather, the only requirement is that all combinations of the 512 addresses be cycled through before being repeated. Equation 1 provides the state definition of the 9-bit refresh address counter and can be used to determine the exact counting sequence. Figure 8 illustrates the gate logic used to create such a counter.

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDRAM: M6 M5 M4 M3 M2 M1 M0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

bits 0–8: Reserved, should be written as a 0 to maintain future compatibility, will be read back as 0.
bits 9–15: M0–M6, are used to define address bits A13–A19 (respectively) of the 20-bit memory refresh address. These bits are set to 0 on RESET.

Figure 6. MDRAM Register Format

Address Bit 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Physical Refresh Address M6 M5 M4 M3 M2 M1 M0 0 0 C A 8 C A 7 C A 6 C A 5 C A 4 C A 3 C A 2 C A 1 C A 0 1

Bit 0: Always driven to a 1 (Logic High). This is true for both the 80C186/80C188.
Bits 1–9: C A 0–C A 9, are generated by the 9-bit Linear-Feedback shift counter.
Bits 10–12: Always driven to a 0 (Logic Low).
Bits 13–19: M0–M6, are defined by the MDRAM Register.

Figure 7. Physical Refresh Address Generation

C A 0 ← C A 1
C A 1 ← C A 2
C A 2 ← C A 3
C A 3 ← C A 4
C A 4 ← C A 5
C A 5 ← C A 6
C A 6 ← if (C A 1 – C A 6 = 1111111B),
then C A 6 = Inverted C A 6
else C A 6 = (((C A 6 .XOR. C A 1) .XNOR. (C A 2 .XOR. C A 3))
C A 7 . C A 8 ← if (C A 0 – C A 6 = 0111111B)
then C A 7 . C A 8 = C A 7 . C A 8 + 1
else C A 7 . C A 8 = C A 7 . C A 8

Equation 1. Refresh Counter Operation
Figure 8. Logic Representation of Refresh Address Counter

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDRAM:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>C8</td>
<td>C7</td>
<td>C6</td>
<td>C5</td>
<td>C4</td>
<td>C3</td>
<td>C2</td>
<td>C1</td>
<td>C0</td>
</tr>
</tbody>
</table>

Bits 0–8: C0–C8, define the number of CLKOUT cycles between each Refresh Request
0 = 512, 1 = 1, 2 = 2, ... These bits are set to 0 on RESET.
Bits 9–15: Reserved, should be written as 0 to maintain future compatibility, will always be read back as 0.

Figure 9. CDRAM Register Format

There are no limitations placed on the programming of the MDRAM register, but be aware that any chip select memory region that overlaps the address established by the MDRAM register will be activated during refresh bus cycles. Therefore, the register should be programmed to correspond to the chip select address that is activated for the dynamic memory partition.

Programming the Refresh Clock Register

The CDRAM register (Figure 9) is used to define the rate at which refresh requests will be internally generated. The CDRAM register is used to maintain the starting value of a down counter, which decrements each falling edge of CLKOUT. When the counter decrements to 1, a refresh request is generated and the counter is again loaded with the value contained in the CDRAM register. Initially, however, the contents of the CDRAM register is loaded into the down counter when the enable bit in the EDRAM register set. Thus, if the CDRAM register is changed, the new value will take effect when either the down counter reaches 1 and reloads itself, or whenever the E bit is written to a 1 (this is true whether the bit was previously set or not). When the 80C186 is reset, the CDRAM register is initialized to zero. A value of zero in the CDRAM register is used to indicate the maximum count rate of 512 clocks.
\[
\text{R}_{\text{PERIOD}} (\mu\text{s}) \times \text{FREQ (MHz)} \over \# \text{Refresh Rows} + (\# \text{Refresh Rows} \times \% \text{Overhead}) = \text{CDRAM Register Value}
\]

\( \text{R}_{\text{PERIOD}} \) = Maximum Refresh period specified by the DRAM manufacturer (time in microseconds).

FREQ = Operating Frequency at 80C186 in megahertz.

\# Refresh Rows = Total number of rows to be refreshed.

\% Overhead = Derating factor that estimates the number of missed refresh requests (typically 1–5%).

**Figure 10. Equation to Calculate Refresh Interval**

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>T8</td>
<td>T7</td>
<td>T6</td>
<td>T5</td>
<td>T4</td>
<td>T3</td>
<td>T2</td>
<td>T1</td>
<td>T0</td>
<td></td>
</tr>
</tbody>
</table>

Bits 0–8: \( T_0 - T_8 \), Refresh request down counter clock count. These bits are read only and represent the current value of the counter. Any write operation to these bits is ignored. These bits are set to 0 on RESET or when the E bit is cleared.

Bits 9–14: Reserved, should be written to as a 0 to maintain future compatibility, will always be read back as zero.

Bit 15: E, enables the operation of the refresh control unit. Setting the E bit will automatically load the Request Down Counter. Clearing the E bit stops refresh operation and clears the Down Counter.

**Figure 11. EDRAM Register Format**

The equation shown in Figure 10 can be used to determine the value of the CDRAM register needed to establish a desired refresh request rate. Note that the equation is based on the internal operating frequency of the 80C186. Therefore, the request rate is effected by any change in operating frequency. Modification of the operating frequency can occur in two ways: modifying the input clock or entering power-save mode. There is no upper limitation as to the frequency of refresh requests (other than programming), but there is a lower limit. This lower limit is based on the fact that the request rate can be no faster than the time it takes to service the request. Subsequently, the minimum programming value of the CDRAM register should be 18 (12H). It is very doubtful that this will ever become a problem when operating at normal frequencies, since the refresh rate of most dynamic memories is well above this minimum programming value.

However, when making use of the power-save feature of the 80C186, it is possible to lower the operating frequency such that it will prevent adequate refreshing rates. When operating at 12.5 MHz, dividing the clock by 16 results in a cycle time of 1.28 microseconds. Since the minimum value of the CDRAM is 18, the minimum refresh rate is 23.04 microseconds. 23 microseconds is not fast enough to service most dynamic memories. Therefore, caution must be exercised when using the power-save feature of the 80C186. When there is a need to keep dynamic memory alive, the clock should not be divided much below 2 MHz to avoid monopolizing the bus with refresh activity. If there is no desire to keep memory alive during power-save operation, then the refresh unit can simply be disabled during this time.

**Programming the Refresh Enable Register**

The EDRAM register (Figure 11) is used to enable and disable the refresh control unit. Furthermore, reading the register returns the current value of the down counter.

Setting the E bit enables the RCU and loads the value of the CDRAM register into the down counter. Whenever the E bit is cleared, the refresh control unit is disabled and the down counter is cleared. Disabling the refresh control unit does not change the contents of the refresh address counter (i.e. it is not cleared or initialized to any specific value). Thus, when the refresh unit is again enabled, the address generated will continue from where it left off. Resetting the 80C186 automatically clears the E bit. There are no refresh bus cycles during a reset.

The current value of the down counter, as well as the present state of the E bit can be examined whenever the EDRAM register is read. Any unused bits will be returned as zero. Whenever the E bit is cleared, the T0 through T8 bits will be read as zero.

**REFRESH CONTROL UNIT OPERATION**

Figure 12 illustrates the two major operational functions of the refresh control unit that are responsible for initiating and controlling DRAM refresh bus cycles.
The down counter is loaded (with the contents of the CDRAM register) on the falling edge of CLKOUT, either when the EFRSH bit is set or whenever the counter decrements to 1. Once loaded, the down counter will decrement every falling edge of CLKOUT. It will continue to decrement as long as the EFRSH bit remains set.

When the down counter finally decrements to 1, two things will happen. First, a request is generated to the BIU to run a refresh bus cycle. The request remains active until the bus cycle is run. Second, the down counter is reloaded with the value contained in the CDRAM register. At this time, the down counter will again begin counting down every clock cycle, it does not wait until the request has been serviced. This is done to ensure that each refresh request occurs at the correct interval. Otherwise, if the down counter only started after the previous request were service, the time between refresh requests would also be a function of bus activity, which for the most part is unpredictable. When the BIU services the refresh request, it will clear the request and increment the refresh address.

**80C188 Address Considerations**

The physical address that is generated during a refresh bus cycle is shown in Figure 7, and it applies to both the 80C186 and 80C188. For the 80C188, this means that the lower address bit A0 will not toggle during refresh operation. Since the 80C188 has an 8-bit external bus, A0 is used as part of memory address decoding. Whereas the 80C186, with its 16-bit external bus, uses A0 (along with BHE) to select memory banks. Therefore, when designing 80C188 memory subsystems it is important not to include A0 as part of the ROW address that is used as a refresh address. Appendix A illustrates Memory Address Multiplexing Techniques that can be applied to the 80C186 and the 80C188.

**MISSING REFRESH REQUESTS**

Under most operating conditions, the frequency of refresh requests is a small percentage of the bus bandwidth. Still, there are several conditions that may prevent a refresh request from being serviced before another request is generated. These conditions include:

1) **LOCKED Bus Cycles**
2) Long Bus accesses (wait states)
3) Bus HOLD

**LOCKED Bus Cycles**

Whenever the bus is LOCKED, the CPU maintains control of the BIU and will not relinquish it until the locked operation is complete. Therefore, internal operations like refresh and DMA are not allowed to execute until the LOCKED instruction has completed. Where this presents the greatest problem is when an instruction such as a move string is executed, and is locked. The move string instruction can take from several clocks to hundreds of thousands of clocks to complete. Obviously anything that takes longer than 512 clocks to complete will always cause a refresh overflow.
Care should be taken not to generate long executing instructions that require bus accesses and are locked. The refresh request interval can be shortened to compensate for missing requests.

**Long Bus Accesses**

The 80C186 does not provide any mechanism to abort or terminate a bus access in the event ready is not returned within a specified amount of time (the 80C186 will infinitely wait for ready). Therefore, if a bus access is in progress when a refresh request is generated, the bus access must complete before the request will be serviced.

**Bus HOLD**

Special consideration is given when a refresh request is generated and the 80C186 is currently being held off the bus due to a HOLD request.

When another bus master has control of the bus, the HLDA signal is kept active as long as the HOLD input remains active. If a refresh request is generated while HOLD is active, the 80C186 will remove (drive inactive) the HLDA signal to indicate to the other bus master that the 80C186 wishes to regain control of the bus (see Figure 13). If, and only if, the HOLD input is removed will the BIU begin to run the refresh bus cycle.

Therefore, it is the responsibility of the system designer to ensure that the 80C186 can regain the bus if a refresh request is signaled. The sequence of HLDA going inactive while HOLD is active can be used to signal a pending refresh request. HOLD need only go inactive for one clock period to allow the refresh bus cycle to be run. If HOLD is again asserted, the 80C186 will give up the bus after the refresh bus cycle has been run (provided there is not another refresh request generated during that time).

**EFFECTS OF MISSING REFRESH REQUESTS**

If a refresh request has not been serviced before another request is generated, the new request is not recorded and is lost. For instance, if the interval between refresh request is 15 microseconds and one request is lost, then the time between two requests will be 30 microseconds when the next request is finally serviced. In this example, missing one request will add 15 μs to the total refresh time. If it is anticipated that refresh requests may be missed (due to programming or system operation), then the refresh request interval should be shortened to allow for missed requests.

Since the BIU is responsible for maintaining the refresh address counter, missing a refresh requests does not imply that refresh addresses are skipped. In fact, an address can never be skipped unless a reset occurs.

**CONCLUSION**

The Internal Refresh Control Unit of the 80C186 and 80C188 helps solve three issues concerning DRAM refreshing: a way to generate periodic refresh requests; a way to generate refresh addresses; a way to simplify DRAM memory controllers. Once a memory controller has been designed to handle the simple tasks of reading and writing the task of refreshing has already been built in.
NOTES:
1. System generates HOLD request.
2. HLDA is returned and 80C186 floats bus/control.
3. Refresh request is generated internal to 80C186.
4. 80C186 lowers (removes) HLDA to signal that it wants the bus back.
5. 80C186 waits until HOLD is lowered (removed) for at least 1 clock cycle (minimum HOLD setup and hold time) to execute the refresh bus cycle. If HOLD is never lowered, the 80C186 will not take over the bus.
6. 80C186 runs the refresh bus cycle.
7. HOLD can be again asserted after the 1 clock duration.
8. The refresh request is cleared after the bus cycle has been executed.
9. If HOLD was again asserted, the 80C186 will immediately relinquish the bus back. If no HOLD occurred, normal CPU operation will resume.

Figure 13. HOLD/HLDA Timing and Refresh Request
## APPENDIX A

TYPICAL DRAM ADDRESS GENERATION CONSIDERATIONS FOR 80C186/80C188

### 80C186 DESIGNS

<table>
<thead>
<tr>
<th>RAM Type</th>
<th>Row Address (A0–AX)</th>
<th>Column Address (A0–AX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K x 1</td>
<td>(128K Bytes)</td>
<td>A1–A8</td>
</tr>
<tr>
<td>16K x 4</td>
<td>(32K Bytes)</td>
<td>A1–A8</td>
</tr>
<tr>
<td>256K x 1</td>
<td>(512K Bytes)</td>
<td>A1–A9</td>
</tr>
<tr>
<td>64K x 4</td>
<td>(128K Bytes)</td>
<td>A1–A8</td>
</tr>
<tr>
<td>1M x 1</td>
<td>(2M Bytes)</td>
<td>A1–A10</td>
</tr>
<tr>
<td>256K x 4</td>
<td>(512K Bytes)</td>
<td>A1–A9</td>
</tr>
</tbody>
</table>

### 80C188 DESIGNS

NOTE:
Address bit A0 can be used in either RAS or CAS addresses, so long as it is not included in any refresh address bits.

### RAM Type

<table>
<thead>
<tr>
<th>RAM Type</th>
<th>RAS Add</th>
<th>CAS Add</th>
<th>Refresh Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K x 1</td>
<td>A0–A7</td>
<td>A0–A7</td>
<td>A0–A6</td>
</tr>
<tr>
<td>16K x 4</td>
<td>A0–A7</td>
<td>A0–A5</td>
<td>A0–A6</td>
</tr>
<tr>
<td>256K x 1</td>
<td>A0–A8</td>
<td>A0–A8</td>
<td>A0–A7</td>
</tr>
<tr>
<td>64K x 4</td>
<td>A0–A7</td>
<td>A0–A7</td>
<td>A0–A7</td>
</tr>
<tr>
<td>1M x 1</td>
<td>A0–A9</td>
<td>A0–A9</td>
<td>A0–A8</td>
</tr>
<tr>
<td>256K x 4</td>
<td>A0–A8</td>
<td>A0–A8</td>
<td>A0–A8</td>
</tr>
</tbody>
</table>
DRAM Refresh/Control with the 80186/80188

STEVE FARRER
APPLICATIONS ENGINEER
In many low-cost 80186/80188 designs, dynamic memory offers an excellent cost/performance advantage. However, DRAM interfacing is often complicated by the need to perform memory refreshing. This application brief describes how to use the Timer and DMA functionality of the 80186/80188 to perform memory refresh.

THEORY OF OPERATION

Dynamic RAM refreshing is accomplished by strobing a ROW address to every ROW of the DRAM within a given period of time. One way to do this is to perform periodic sequential reads to the DRAM using a DMA controller and a Timer. This can be achieved with the 80186/188 by Programming Timer 2 and one of the DMA channels such that the timer generated one DMA cycle approximately every 15 micro-seconds. Please note that this is a single row refresh method and not a burst refresh. Single row refreshing reduces the bus overhead considerably when compared to burst refreshing.

The control logic of the DRAM is such that a RAS (row address strobe) occurs on every memory read, regardless of the address. This is necessary because the DMA channel is cycling through the entire 1 MByte address space and the address of the refresh cycle does not always fall within the range of the DRAM bank.

Although the address may be outside the DRAM range, the lower address bits continue to change and roll over to provide the row address.

READY LOGIC WITH MEMORY

Since the DMA controller is cycling through the entire 1 MByte address space, care must be taken to ensure that a READY signal is available for all addresses. One way to do this is to use only the internal wait state generator for memory areas and to strap the SRDY and ARDY pins HIGH. Whenever a refresh cycle occurs outside of a predefined internal wait state area, the external ready pins, which are active HIGH, will complete the bus cycle.

If it is necessary to use the external ready signals for certain memory regions, then it will be necessary to add logic which will generate a ready signal whenever the address of a refresh cycle falls where there is no memory. This can easily be accomplished by either decoding a couple of high order address lines, or by AND-ing all the chip selects so that READY goes active whenever all the memory chip selects are inactive (i.e. the cycle is not in a valid memory region).

BUS OVERHEAD

The absolute maximum overhead can be calculated at a given speed by taking the number of refresh cycles divided by the total number of bus cycles for a given period of time. At 8 MHz these values can be calculated as follows:

\[
\frac{2 \text{ bus cycles}}{15.2 \mu s/500 \text{ ns}} \times 100 = 6.6\% \text{ maximum overhead}
\]

In reality, the bus overhead associated with the DMA cycles is much lower due to the instruction prefetch queue. When a DMA cycle is requested by the timer for a refresh cycle, the Bus Interface Unit honors the request on the next bus cycle boundary (with the exception of LOCKed bus cycles and odd aligned accesses). Typically this time is idle time on the bus and the impact on the overall performance is extremely small. The following table shows more realistic data which was acquired by running 6 different benchmarks with and without the DMA channel enabled to provide refresh every 15.2\( \mu \)s.

<table>
<thead>
<tr>
<th>BENCHMARK RESULTS @ 8 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>80186</td>
</tr>
<tr>
<td>80188</td>
</tr>
</tbody>
</table>

The programs which showed the highest bus overhead tended to be very bus intensive. Also note that at faster frequencies the bus overhead becomes even less.

DMA OPERATION

The DMA controller is programmed to be source synchronized with the TC (transfer count) bit cleared. This ensures that the DMA controller never reaches a final count. The source pointer continues to increment through memory on every cycle. When FFFFFH is reached, the address rolls over to 00000H.

The programming values for the DNA registers are shown in Figure 1. The source pointer may be initialized to any location since the starting location of the refresh is arbitrary.
The value of the Transfer Count register is also arbitrary since the TC bit is not set. The DMA channel will continue to run cycles upon request from Timer 2 even after the Transfer Count register has reached zero. Once zero is reached, the Transfer Count register will roll over to $FFFFH$ and continue to count down.

The destination pointer may be set to any available memory or I/O location. This pointer must be set so that it neither increments nor decrements. Otherwise, the address of the deposit cycle would cycle through memory or I/O doing writes which could possibly be destructive. Thus the INC and DEC bits of the control register should be cleared.

### NOTES:
1. Locations of registers are relative to the base address of the peripheral control block. The offsets shown are for Channel 0.
2. The byte/word bit is a don't care in a 80188 system. In a 80186 system this bit should be set to a 1 to represent word transfers.
3. The transfer count register is located at offset C8H. It is not necessary to program this register.

**Figure 1. DMA Registers**
TIMER OPERATION

Timer 2 must be programmed to generate a DMA request every time a row must be refreshed. Since we are not using a burst refresh, the refresh time is divided up evenly among the number of rows. For a 2 ms refresh DRAM with 128 rows, the time between rows equals 15.62 microseconds.

When setting the count value of the timer, keep in mind the timer clock is operating at one-fourth the CPU clock frequency. Thus, the equation for setting the timer count is:

\[
\frac{(\text{CPU CLOUT FREQ}) \times (\text{Time Between ROWS})}{4} = \text{COUNT VALUE (decimal)}
\]

For an 8 MHz clock, programming the Maximum Count Register to 1EH provides a 15.2 µs refresh. This programming is indicated in Figure 2.

![Figure 2. Timer 2 Registers Programmed for a 15.2 µs Refresh at 8 MHz](image-url)
EXAMPLE 1: 
DRAM CONTROL WITH A DELAY LINE

This is the most straightforward way of implementing the RAS and CAS logic. A RAS signal is generated by either RD or WR going active while the address is within the corresponding range. Normally the logic for RAS would also go active for a refresh cycle status, but since this information is not available on the 80186/80188, a RAS must be generated for every RD and WR, regardless address.

The MUX signal is used to change from the RAS address to the CAS address after latching with RAS. This is accomplished by using a delay line which generates a MUX signal by a fixed number of nano-seconds after RAS is generated. The important timing here is the necessary hold time for the row address into the DRAM.

The MUX signal is initially HIGH which sends the A side (see Figure 3) Row address through the multiplexer to the DRAM. This address consists of A0 through A7. The B address (A8 through A16) is selected when MUX goes LOW. The system shown in Figure 3 represents that of an 80188 system.

For an 80186 system, the A address would start at A1. The least significant address line A0 along with BHE would be used to decode WE into WEH and WEL which will be shown in the second example. Also, the 186 DMA must be set to do word transfers so that the address is incremented by 2 after each refresh cycle. This is necessary to ensure A1 increments by 1 every refresh cycle.

CAS is generated in the same manner by delaying the MUX signal a fixed number of nano-seconds. Typically CAS goes inactive at the same time as RAS to ensure a valid CAS precharge time before the next DRAM access. The 80186/188 chip selects are used to ensure that CAS only goes active when the address falls within the DRAM bank range, and to ensure that CAS does not go active during I/O cycles.

Figure 3. Using A Delay Line for DRAM Control
EXAMPLE 2: DRAM CONTROL WITH A PAL*

This design uses a PAL to generate all the control logic for the DRAM array. Internal feedback is used on the signals to control the timing and states of the RAS, MUX and CAS signals.

This design uses 256k X 4 DRAMs. With minor changes to the PAL equations this design could just as easily make use of 64k X 1, 64k X 4, or 256k X 1 DRAMs.

The RAS signal is generated off ALE going LOW, bus cycle status active, and PRE_RAS being active. The PRE_RAS signal is necessary to ensure that a RAS is not accidentally generated when S2-S0 are becoming valid and ALE has not yet gone HIGH in T4 phase 2. PRE_RAS does not go active until ALE has gone HIGH.

RAS is initiated for every memory read and write regardless of the bus cycle address. This ensures a row refresh when the refresh address falls outside of the DRAM bank and also a refresh to both banks simultaneously so that the frequency of the refresh can be set for the number of rows in one bank of DRAM.

The UCS (Upper Chip Select) from the 80186/188 is used to disable DRAM signals when the processor is attempting to access upper memory control ROM. Thus the portion of memory used by the UCS (maximum 256k) is unavailable in the upper DRAM. However, the RAS signal must still be allowed during UCS access to ensure refreshing when the DMA refresh cycle occurs in the UCS region.

MUX is generated off T2 phase 1 and RAS active. MUX will remain low until the current RAS signal goes inactive during T3 phase 2.

CAS0 and CAS1 are generated off MUX being active and T2 phase 2 of the bus cycle. CAS goes inactive at the start of T4 phase 2.

---

*PAL® is a registered trademark of Monolithic Memories.
Figure 5. Timing Diagram for PAL DRAM Controller
The following equations are with reference to given clock edge. The edge in reference is indicated by the first element in the equation: \( T_3 \uparrow = \) rising edge of \( T_3 \) clock \( \downarrow T_1 = \) falling edge of \( T_1 \) clock.

\[
\begin{align*}
\text{DELAY 1} &= T_1 \uparrow + T_{CHLL} + (\text{PAL DELAY}) \\
\text{DELAY 2} &= \downarrow T_2 + (\text{PAL DELAY}) \\
\text{DELAY 3} &= T_2 \uparrow + (\text{PAL DELAY}) \\
\text{DELAY 4} &= \downarrow T_1 + (\text{PAL DELAY}) \\
\text{DELAY 5} &= \downarrow T_3 + T_{CLSH} + (\text{PAL DELAY}) \\
\text{DELAY 6} &= \downarrow T_1 + T_{CLAV} + (\text{MUX DELAY}) \\
\text{DELAY 7} &= \downarrow T_2 + \text{DELAY 2} + (\text{MUX DELAY})
\end{align*}
\]

\[
\begin{align*}
\text{ACCESS TIME FROM RAS} &= 2.5 \text{ (TCLCL)} - \text{DELAY 1} - \text{TDVCL} \\
\text{ACCESS TIME FROM CAS} &= 1.5 \text{ (TCLCL)} - \text{DELAY 3} - \text{TDVCL}
\end{align*}
\]
Using The 8096

IRA HORDEN
MCO APPLICATIONS ENGINEER
1.0 INTRODUCTION

High speed digital signals are frequently encountered in modern control applications. In addition, there is often a requirement for high speed 16-bit and 32-bit precision in calculations. The MCS®-96 product line, generically referred to as the 8096, is designed to be used in applications which require high speed calculations and fast I/O operations.

The 8096 is a 16-bit microcontroller with dedicated I/O subsystems and a complete set of 16-bit arithmetic instructions including multiply and divide operations. This Ap-note will briefly describe the 8096 in section 2, and then give short examples of how to use each of its key features in section 3. The concluding sections feature a few examples which make use of several chip features simultaneously and some hardware connection suggestions. Further information on the 8096 and its use is available from the sources listed in the bibliography.

2.0 8096 OVERVIEW

2.1. General Description

Unlike microprocessors, microcontrollers are generally optimized for specific applications. Intel's 8048 was optimized for general control tasks while the 8051 was optimized for 8-bit math and single bit boolean operations. The 8096 has been designed for high speed/high performance control applications. Because it has been designed for these applications the 8096 architecture is different from that of the 8048 or 8051.

There are two major sections of the 8096; the CPU section and the I/O section. Each of these sections can be subdivided into functional blocks as shown in Figure 2-1.

![8096 Block Diagram](image-url)
2.1.1. CPU SECTION

The CPU of the 8096 uses a 16-bit ALU which operates on a 256-byte register file instead of an accumulator. Any of the locations in the register file can be used for sources or destinations for most of the instructions. This is called a register to register architecture. Many of the instructions can also use bytes or words from anywhere in the 64K byte address space as operands. A memory map is shown in Figure 2-2.

In the lower 24 bytes of the register file are the register-mapped I/O control locations, also called Special Function Registers or SFRs. These registers are used to control the on-chip I/O features. The remaining 232 bytes are general purpose RAM, the upper 16 of which can be kept alive using a low current power-down mode.

<table>
<thead>
<tr>
<th>65535</th>
<th>EXTERNAL MEMORY OR I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384</td>
<td></td>
</tr>
<tr>
<td>8320</td>
<td>INTERNAL PROGRAM STORAGE ROM</td>
</tr>
<tr>
<td>8210</td>
<td>FACTORY TEST CODE</td>
</tr>
<tr>
<td>8192</td>
<td>INTERRUPT 8</td>
</tr>
<tr>
<td>8190</td>
<td>VECTORS 0</td>
</tr>
<tr>
<td>256</td>
<td>PORT 4</td>
</tr>
<tr>
<td>255</td>
<td>PORT 3</td>
</tr>
<tr>
<td>00</td>
<td>EXTERNAL MEMORY OR I/O</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0000H</td>
<td>INTERNAL RAM REGISTER FILE</td>
</tr>
<tr>
<td></td>
<td>STACK POINTER</td>
</tr>
<tr>
<td></td>
<td>SPECIAL FUNCTION REGISTERS (WHEN ACCESSED AS DATA MEMORY)</td>
</tr>
<tr>
<td>0100H</td>
<td></td>
</tr>
<tr>
<td>0FFFH</td>
<td>EXTERNAL MEMORY RESERVED FOR USE BY INTEL DEVELOPMENT SYSTEMS (WHEN ACCESSED AS PROGRAM MEMORY)</td>
</tr>
<tr>
<td>FFFFH</td>
<td></td>
</tr>
<tr>
<td>4000H</td>
<td></td>
</tr>
<tr>
<td>2080H -- RESET</td>
<td></td>
</tr>
<tr>
<td>2012H</td>
<td></td>
</tr>
<tr>
<td>2000H</td>
<td></td>
</tr>
<tr>
<td>0FFEH</td>
<td></td>
</tr>
<tr>
<td>0000H</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-2. Memory Map
Figure 2-3 shows the layout of the register mapped I/O. Some of these registers serve two functions, one if they are read from and another if they are written to. More information about the use of these registers is included in the description of the features which they control.

Figure 2-3: SFR Layout
2.1.2. I/O FEATURES

Many of the I/O features on the 8096 are designed to operate with little CPU intervention. A list of the major I/O functions is shown in Figure 2-4. The Watchdog Timer is an internal timer which can be used to reset the system if the software fails to operate properly. The Pulse-Width-Modulation (PWM) output can be used as a rough D to A, a motor driver, or for many other purposes. The A to D converter (ADC) has 8 multiplexed inputs and 10-bit resolution. The serial port has several modes and its own baud rate generator. The High Speed I/O section includes a 16-bit timer, a 16-bit counter, a 4-input programmable edge detector, 4 software timers, and a 6-output programmable event generator. All of these features will be described in section 2.3.

2.2. The Processor Section

2.2.1. OPERATIONS AND ADDRESSING MODES

The 8096 has 100 instructions, some of which operate on bits, some on bytes, some on words and some on longs (double words). All of the standard logical and arithmetic functions are available for both byte and word operations. Bit operations and long operations are provided for some instructions. There are also flag manipulation instructions as well as jump and call instructions. A full set of conditional jumps has been included to speed up testing for various conditions.

Bit operations are provided by the Jump Bit and Jump Not Bit instructions, as well as by immediate masking of bytes. These bit operations can be performed on any of the bytes in the register file or on any of the special function registers. The fast bit manipulation of the SFRs can provide rapid I/O operations.

A symmetric set of byte and word operations make up the majority of the 8096 instruction set. The assembly language for the 8096 (ASM-96) uses a “B” suffix on a mnemonic to indicate a byte operation, without this suffix a word operation is indicated. Many of these operations can have one, two or three operands. An example of a one operand instruction would be:

```
NOT Value1 ; Value1 = 1's complement (Value1)
```

A two operand instruction would have the form:

```
ADD Value2,Value1 ; Value2 = Value2 + Value1
```

A three operand instruction might look like:

```
MUL Value3,Value2,Value1 ; Value3 = Value2\^\* Value1
```

The three operand instructions combined with the register to register architecture almost eliminate the necessity of using temporary registers. This results in a faster processing time than machines that have equivalent instruction execution times, but use a standard architecture.

Long (32-bit) operations include shifts, normalize, and multiply and divide. The word divide is a 32-bit by 16-bit operation with a 16-bit quotient and 16-bit remainder. The word multiply is a word by word multiply with a long result. Both of these operations can be done in either the signed or unsigned mode. The direct unsigned modes of these instructions take only 6.5 microseconds. A normalize instruction and sticky bit flag have been included in the instruction set to provide hardware support for the software floating point package (FPAL-96).

<table>
<thead>
<tr>
<th>Major I/O Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Speed Input Unit</td>
<td>Provides Automatic Recording of Events</td>
</tr>
<tr>
<td>High Speed Output Unit</td>
<td>Provides Automatic Triggering of Events and Real-Time Interrupts</td>
</tr>
<tr>
<td>Pulse Width Modulation</td>
<td>Output to Drive Motors or Analog Circuits</td>
</tr>
<tr>
<td>A to D Converter</td>
<td>Provides Analog Input</td>
</tr>
<tr>
<td>Watchdog Timer</td>
<td>Resets 8096 if a Malfunction Occurs</td>
</tr>
<tr>
<td>Serial Port</td>
<td>Provides Synchronous or Asynchronous Link</td>
</tr>
<tr>
<td>Standard I/O Lines</td>
<td>Provide Interface to the External World when other Special Features are not needed</td>
</tr>
</tbody>
</table>

Figure 2-4. Major I/O Functions
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Operation (Note 1)</th>
<th>Flags</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD/ADDB</td>
<td>2</td>
<td>D ← D + A</td>
<td>Z N C V VT ST</td>
<td></td>
</tr>
<tr>
<td>ADD/ADDB</td>
<td>3</td>
<td>D ← B + A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDC/ADDCB</td>
<td>2</td>
<td>D ← D + A + C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB/SUBB</td>
<td>2</td>
<td>D ← D - A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB/SUBB</td>
<td>3</td>
<td>D ← D - B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBC/SUBCB</td>
<td>2</td>
<td>D ← D - A + C - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP/CMPB</td>
<td>2</td>
<td>D ← A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL/MULU</td>
<td>2</td>
<td>D, D + 2 ← D * A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL/MULU</td>
<td>3</td>
<td>D, D + 2 ← B * A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULB/MULUB</td>
<td>2</td>
<td>D, D + 1 ← D * A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULB/MULUB</td>
<td>3</td>
<td>D, D + 1 ← B * A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVU</td>
<td>2</td>
<td>D ← (D, D + 2) / A, D + 2 ← remainder</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>DIVUB</td>
<td>2</td>
<td>D ← (D, D + 1) / A, D + 1 ← remainder</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>DIV</td>
<td>2</td>
<td>D ← (D, D + 2) / A, D + 2 ← remainder</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>DIVB</td>
<td>2</td>
<td>D ← (D, D + 1) / A, D + 1 ← remainder</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>AND/ANDB</td>
<td>2</td>
<td>D ← D and A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND/ANDB</td>
<td>3</td>
<td>D ← B and A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR/ORB</td>
<td>2</td>
<td>D ← D or A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR/XORB</td>
<td>2</td>
<td>D ← D (excl. or) A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD/LDB</td>
<td>2</td>
<td>D ← A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST/STB</td>
<td>2</td>
<td>A ← D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDBSE</td>
<td>2</td>
<td>D ← A; D + 1 ← SIGN(A)</td>
<td></td>
<td>3,4</td>
</tr>
<tr>
<td>LDBZE</td>
<td>2</td>
<td>D ← A; D + 1 ← 0</td>
<td></td>
<td>3,4</td>
</tr>
<tr>
<td>PUSH</td>
<td>1</td>
<td>SP ← SP - 2; (SP) ← A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>1</td>
<td>A ← (SP); SP ← SP + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSHF</td>
<td>0</td>
<td>SP ← SP - 2; (SP) ← PSW; PSW ← 0000H</td>
<td>0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td>0</td>
<td>PSW ← (SP); SP ← SP + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SJMP</td>
<td>1</td>
<td>PC ← PC + 11-bit offset</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>LJMP</td>
<td>1</td>
<td>PC ← PC + 16-bit offset</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>BR (indirect)</td>
<td>1</td>
<td>PC ← (A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCALL</td>
<td>1</td>
<td>SP ← SP - 2; (SP) ← PC; PC ← PC + 11-bit offset</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>SCALL</td>
<td>1</td>
<td>SP ← SP - 2; (SP) ← PC; PC ← PC + 11-bit offset</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>LCALL</td>
<td>1</td>
<td>SP ← SP - 2; (SP) ← PC; PC ← PC + 16-bit offset</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>RET</td>
<td>0</td>
<td>PC ← (SP); SP ← SP + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J (conditional)</td>
<td>1</td>
<td>PC ← PC + 8-bit offset (if taken)</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JC</td>
<td>1</td>
<td>Jump if C = 1</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JNC</td>
<td>1</td>
<td>Jump if C = 0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JE</td>
<td>1</td>
<td>Jump if Z = 1</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

**Figure 2-5. Instruction Summary**

**NOTES:**
1. If the mnemonic ends in "B", a byte operation is performed, otherwise a word operation is done. Operands D, B, and A must conform to the alignment rules for the required operand type. D and B are locations in the register file; A can be located anywhere in memory.
2. D, D + 2 are consecutive WORDS in memory; D is DOUBLE-WORD aligned.
3. D, D + 1 are consecutive BYTES in memory; D is WORD aligned.
4. Changes a byte to a word.
5. Offset is a 2's complement number.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Operation (Note 1)</th>
<th>Flags</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNE</td>
<td>1</td>
<td>Jump if Z = 0</td>
<td>Z N C V VT ST</td>
<td>5</td>
</tr>
<tr>
<td>JGE</td>
<td>1</td>
<td>Jump if N = 0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JLT</td>
<td>1</td>
<td>Jump if N = 1</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JGT</td>
<td>1</td>
<td>Jump if N = 0 and Z = 0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JLE</td>
<td>1</td>
<td>Jump if N = 1 or Z = 1</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JH</td>
<td>1</td>
<td>Jump if C = 1 and Z = 0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JNH</td>
<td>1</td>
<td>Jump if C = 0 or Z = 1</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JV</td>
<td>1</td>
<td>Jump if V = 1</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JNV</td>
<td>1</td>
<td>Jump if V = 0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JVT</td>
<td>1</td>
<td>Jump if VT = 1; Clear VT</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JNVT</td>
<td>1</td>
<td>Jump if VT = 0; Clear VT</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JST</td>
<td>1</td>
<td>Jump if ST = 1</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JNST</td>
<td>1</td>
<td>Jump if ST = 0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>JBS</td>
<td>3</td>
<td>Jump if Specified Bit = 1</td>
<td></td>
<td>5,6</td>
</tr>
<tr>
<td>JBC</td>
<td>3</td>
<td>Jump if Specified Bit = 0</td>
<td></td>
<td>5,6</td>
</tr>
<tr>
<td>DJNZ</td>
<td>1</td>
<td>D ↔ D - 1; if D ≠ 0 then PC ↔ PC + 8-bit offset</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>DEC/DECB</td>
<td>1</td>
<td>D ↔ D - 1</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>2</td>
</tr>
<tr>
<td>NEG/NEGB</td>
<td>1</td>
<td>D ↔ 0 - D</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>3</td>
</tr>
<tr>
<td>INC/INCB</td>
<td>1</td>
<td>D ↔ D + 1</td>
<td>✓ ✓ ✓ ✓ ✓</td>
<td>7</td>
</tr>
<tr>
<td>EXT</td>
<td>1</td>
<td>D ↔ D; D + 2 ↔ Sign (D)</td>
<td>✓ ✓ 0 0 — —</td>
<td></td>
</tr>
<tr>
<td>EXTB</td>
<td>1</td>
<td>D ↔ D; D + 1 ↔ Sign (D)</td>
<td>✓ ✓ 0 0 — —</td>
<td></td>
</tr>
<tr>
<td>NOT/NOTB</td>
<td>1</td>
<td>D ↔ Logical Not (D)</td>
<td>✓ ✓ 0 0 — —</td>
<td></td>
</tr>
<tr>
<td>CLR/CLRB</td>
<td>1</td>
<td>D ↔ 0</td>
<td>1 0 0 0 — —</td>
<td></td>
</tr>
<tr>
<td>SHL/SHLB/SHLL</td>
<td>2</td>
<td>C ↔ msb — — — — — — Isb ↔ 0</td>
<td>✓ ? ✓ ✓ ✓ ✓</td>
<td>7</td>
</tr>
<tr>
<td>SHR/SHRB/SHRL</td>
<td>2</td>
<td>0 ↔ msb — — — — — — Isb → C</td>
<td>✓ ? ✓ ✓ 0 — ✓</td>
<td>7</td>
</tr>
<tr>
<td>SHRA/SHRAB/SHRAL</td>
<td>2</td>
<td>msb ↔ msb — — — — — — Isb → C</td>
<td>✓ ✓ ✓ ✓ 0 — ✓</td>
<td>7</td>
</tr>
<tr>
<td>SETC</td>
<td>0</td>
<td>C ↔ 1</td>
<td>0 0 1 — — —</td>
<td></td>
</tr>
<tr>
<td>CLRC</td>
<td>0</td>
<td>C ↔ 0</td>
<td>0 0 0 — — —</td>
<td></td>
</tr>
<tr>
<td>CLRVT</td>
<td>0</td>
<td>VT ↔ 0</td>
<td>0 0 0 0 — 0</td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td>0</td>
<td>PC ↔ 2080H</td>
<td>0 0 0 0 0 0 0 0 8</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>0</td>
<td>Disable All Interrupts (I ↔ 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EI</td>
<td>0</td>
<td>Enable All Interrupts (I ↔ 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>0</td>
<td>PC ↔ PC + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SKIP</td>
<td>0</td>
<td>PC ↔ PC + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NORML</td>
<td>2</td>
<td>Left Shift Till msb = 1; D ↔ shift count</td>
<td>✓ ? 0 — — — —</td>
<td>7</td>
</tr>
<tr>
<td>TRAP</td>
<td>0</td>
<td>SP ↔ SP - 2; (SP) ↔ PC</td>
<td>— — — — — — — —</td>
<td>9</td>
</tr>
</tbody>
</table>

**Figure 2-5. Instruction Summary (Continued)**

**NOTES:**
1. If the mnemonic ends in "B", a byte operation is performed, otherwise a word operation is done. Operands D, B, and A must conform to the alignment rules for the required operand type. D and B are locations in the register file; A can be located anywhere in memory.
2. Offset is a 2's complement number.
3. Specified bit is one of the 2048 bits in the register file.
4. The "L" (Long) suffix indicates double-word operation.
5. Initiates a Reset by pulling RESET low. Software should re-initialize all the necessary registers with code starting at 2080H.
6. The assembler will not accept this mnemonic.
One operand of most of the instructions can be used with any one of six addressing modes. These modes increase the flexibility and overall execution speed of the 8096. The addressing modes are: register-direct, immediate, indirect, indirect with auto-increment, and long and short indexed.

The fastest instruction execution is gained by using either register direct or immediate addressing. Register-direct addressing is similar to normal direct addressing, except that only addresses in the register file or SFRs can be addressed. The indexed mode is used to directly address the remainder of the 64K address space. Immediate addressing operates as would be expected, using the data following the opcode as the operand.

Both of the indirect addressing modes use the value in a word register as the address of the operand. If the indirect auto-increment mode is used then the word register is incremented by one after a byte access or by two after a word access. This mode is particularly useful for accessing lookup tables.

Access to any of the locations in the 64K address space can be obtained by using the long indexed addressing mode. In this mode a 16-bit 2’s complement value is added to the contents of a word register to form the address of the operand. By using the zero register as the index, ASM96 (the assembler) can accept “direct” addressing to any location. The zero register is located at 0000H and always has a value of zero. A short indexed mode is also available to save some time and code. This mode uses an 8-bit 2’s complement number as the offset instead of a 16-bit number.

### 2.2.2. ASSEMBLY LANGUAGE

The multiple addressing modes of the 8096 make it easy to program in assembly language and provide an excellent interface to high level languages. The instructions accepted by the assembler consist of mnemonics followed by either addresses or data. A list of the mnemonics and their functions are shown in Figure 2-5. The addresses or data are given in different formats depending on the addressing mode. These modes and formats are shown in Figure 2-6.

Additional information on 8096 assembly language is available in the MCS-96 Macro Assembler Users Guide, listed in the bibliography.

<table>
<thead>
<tr>
<th>Mnem</th>
<th>Dest or Src1</th>
<th>One operand direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mnem</td>
<td>Dest, Src1</td>
<td>Two operand direct</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, Src1, Src2</td>
<td>Three operand direct</td>
</tr>
<tr>
<td>Mnem</td>
<td>#Src1</td>
<td>One operand immediate</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, #Src1</td>
<td>Two operand immediate</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, #Src1, #Src2</td>
<td>Three operand immediate</td>
</tr>
<tr>
<td>Mnem</td>
<td>[addr]</td>
<td>One operand indirect</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, [addr]</td>
<td>Two operand indirect</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, [addr]</td>
<td>Two operand indirect auto-increment</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, [addr]</td>
<td>Two operand indirect auto-increment</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, Src1, [addr]</td>
<td>Three operand indirect</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, Src1, [addr]</td>
<td>Three operand indirect auto-increment</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, offs [addr]</td>
<td>Two operand indexed (short or long)</td>
</tr>
<tr>
<td>Mnem</td>
<td>Dest, Src1, offs [addr]</td>
<td>Three operand indexed (short or long)</td>
</tr>
</tbody>
</table>

Where: “Mnem” is the instruction mnemonic
“Dest” is the destination register
“Src1”, “Src2” are the source registers
“addr” is a register containing a value to be used in computing the address of an operand
“offs” is an offset used in computing the address of an operand

**Figure 2-6. Instruction Format**
2.2.3. INTERRUPTS

The flexibility of the instruction set is carried through into the interrupt system. There are 20 different interrupt sources that can be used on the 8096. The 20 sources vector through 8 locations or interrupt vectors. The vector names and their sources are shown in Figure 2-7, with their locations listed in Figure 2-8. Control of the interrupts is handled through the Interrupt Pending Register (INT_PENDING), the Interrupt Mask Register (INT_MASK), and the I bit in the PSW (PSW.9). Figure 2-9 shows a block diagram of the interrupt structure. The INT_PENDING register contains bits which get set by hardware when an interrupt occurs. If the interrupt mask register bit for that source is a 1 and PSW.9 = 1, a vector will be taken to the address listed in the interrupt vector table for that

<table>
<thead>
<tr>
<th>Source</th>
<th>Vector Location (High Byte)</th>
<th>Vector Location (Low Byte)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>2011H</td>
<td>2010H</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>Extint</td>
<td>200FH</td>
<td>200EH</td>
<td>7 (Highest)</td>
</tr>
<tr>
<td>Serial Port</td>
<td>200DH</td>
<td>200CH</td>
<td>6</td>
</tr>
<tr>
<td>Software Timers</td>
<td>200BH</td>
<td>200AH</td>
<td>5</td>
</tr>
<tr>
<td>HSI.0</td>
<td>2009H</td>
<td>2008H</td>
<td>4</td>
</tr>
<tr>
<td>High Speed</td>
<td>2007H</td>
<td>2006H</td>
<td>3</td>
</tr>
<tr>
<td>Outputs</td>
<td>2005H</td>
<td>2004H</td>
<td>2</td>
</tr>
<tr>
<td>HSI Data Available</td>
<td>2003H</td>
<td>2002H</td>
<td>1</td>
</tr>
<tr>
<td>A/D Conversion Complete</td>
<td>2001H</td>
<td>2000H</td>
<td>0 (Lowest)</td>
</tr>
</tbody>
</table>

Figure 2-8. Interrupt Vectors and Priorities
source. When the vector is taken the INT>PENDING bit is cleared. If more than one bit is set in the INT>PENDING register with the corresponding bit set in the INT>MASK register, the Interrupt with the highest priority shown in Figure 2-8 will be executed.

The software can make the hardware interrupts work in almost any fashion desired by having each routine run with its own setup in the INT>MASK register. This will be clearly seen in the examples in section 4 which change the priority of the vectors in software. The

![Diagram of Interrupt Structure Block Diagram](image-url)
**Figure 2-10. The PSW Register**

PSW (shown in Figure 2-10), stores the INT_MASK register in its lower byte so that the mask register can be pushed and popped along with the machine status when moving in and out of routines. The action of pushing flags clears the PSW which includes PSW.9, the interrupt enable bit. Therefore, after a PUSHF instruction interrupts are disabled. In most cases an interrupt service routine will have the basic structure shown below.

**INT VECTOR:**

```plaintext
PUSHF
LDB INT_MASK, #xxxxxxxxXB
EI
- ;Insert service routine here
- 
PPOP
RET
```

The PUSHF instruction saves the PSW including the old INT_MASK register. The PSW, including the interrupt enable bit are left cleared. If some interrupts need to be enabled while the service routine runs, the INT_MASK is loaded with a new value and interrupts are globally enabled before the service routine continues. At the end of the service routine a POPF instruction is executed to restore the old PSW. The RET instruction is executed and the code returns to the desired location. Although the PPOP instruction can enable the interrupts the next instruction will always execute. This prevents unnecessary building of the stack by ensuring that the RET always executes before another interrupt vector is taken.

### 2.3. On-Chip I/O Section

All of the on-chip I/O features of the 8096 can be accessed through the special function registers, as shown in Figure 2-3. The advantage of using register-mapped I/O is that these registers can be used as the sources or destinations of CPU operations. There are seven major I/O functions. Each one of these will be considered with a section of code to exemplify its usage. The first section covered will be the High Speed I/O, (HSIO), subsystem. This section includes the High Speed Input (HSI) unit, High Speed Output (HSO) unit, and the Timer/Counter section.

#### 2.3.1. TIMER/COUNTERS

The 8096 has two time bases, Timer 1 and Timer 2. Timer 1 is a 16-bit free running timer which is incremented every 8 state times. (A state time is 3 oscillator periods, or 0.25 microseconds with a 12 MHz crystal.)
Its value can be read at any time and used as a reference for both the HSI section and the HSO section. Timer 1 can cause an interrupt when it overflows, and cannot be modified or stopped without resetting the entire chip. Timer 2 is really an event counter since it uses an external clock source. Like Timer 1, it is 16-bits wide, can be read at any time, can be used with the HSO section, and can generate an interrupt when it overflows. Control of Timer 2 is limited to incrementing it and resetting it. Specific values can not be written to it.

Although the 8096 has only two timers, the timer flexibility is equal to a unit with many timers thanks to the HSIO unit. The HSI enables one to measure times of external events on up to four lines using Timer 1 as a timer base. The HSO unit can schedule and execute internal events and up to six external events based on the values in either Timer 1 or Timer 2. The 8096 also includes separate, dedicated timers for the baud rate generator and watchdog timer.

2.3.2. HSI

The HSI unit can be thought of as a message taker which records the line which had an event and the time at which the event occurred. Four types of events can trigger the HSI unit, as shown in the HSI block diagram in Figure 2-11. The HSI unit can measure pulse widths and record times of events with a 2

![Figure 2-11. HSI Unit Block Diagram](image)

- Pulse measurement with 2.0 μsec resolution
- Input transitions trigger the recording of the reference
  Timer (16-bit) and triggered input(s) (4-bit)

![Figure 2-12. HSI Mode Register](image)

Where each 2-bit mode control field defines one of 4 possible modes:
- 00  8 positive transitions
- 01  Each positive transition
- 10  Each negative transition
- 11  Every transition (positive and negative)
microsecond resolution. It can look for one of four events on each of four lines simultaneously, based on the information in the HSI Mode register, shown in Figure 2-12. The information is then stored in a seven level FIFO for later retrieval. Whenever the FIFO contains information, the earliest entry is placed in the holding register. When the holding register is read, the next valid piece of information is loaded into it. Interrupts can be generated by the HSI unit at the time the holding register is loaded or when the FIFO has six or more entries.

2.3.3. HSO

Just as the HSI can be thought of as a message taker, the HSO can be thought of as a message sender. At times determined by the software, the HSO sends mes-

---

Figure 2-13. HSO Command Register

Figure 2-14. HSO Block Diagram
sages to various devices to have them turn on, turn off, start processing, or reset. Since the programmed times can be referenced to either Timer 1 or Timer 2, the HSO makes the two timers look like many. For example, if several events have to occur at specific times, the HSO unit can schedule all of the events based on a single timer. The events that can be scheduled to occur and the format of the command written to the HSO Command register are shown in Figure 2-13.

The software timers listed in the figure are actually 4 software flags in I/O Status Register 1 (I0S1). These flags can be set, and optionally cause an interrupt, at any time based on Timer 1 or Timer 2. In most cases these timers are used to trigger interrupt routines which must occur at regular intervals. A multitask process can easily be set up using the software timers.

A CAM (Content Addressable Memory) file is the main component of the HSO. This file stores up to eight events which are pending to occur. Every state time one location of the CAM is compared with the two timers. After 8 state times, (two microseconds with a 12 MHz clock), the entire CAM has been searched for time matches. If a match occurs the specified event will be triggered and that location of the CAM will be made available for another pending event. A block diagram of the HSO unit is shown in Figure 2-14.

2.3.4. Serial Port

Controlling a device from a remote location is a simple task that frequently requires additional hardware with many processors. The 8096 has an on-chip serial port to reduce the total number of chips required in the system.
The serial port is similar to that on the MCS-51 product line. It has one synchronous and three asynchronous modes. In the asynchronous modes baud rates of up to 187.5 Kbaud can be used, while in the synchronous modes rates up to 1.5 Mbaud are available. The chip has a baud rate generator which is independent of Timer 1 and Timer 2, so using the serial port does not take away any of the HSI, HSO or timer flexibility or functionality.

Control of the serial port is provided through the SPCON/SPSTAT (Serial Port CONtrol/Serial Port STATus) register. This register, shown in Figure 2-15, has some bits which are read only and others which are write only. Although the functionality of the port is similar to that of the 8051, the names of some of the modes and control bits are different. The way in which the port is used from a software standpoint is also slightly different since RI and TI are cleared after each read of the register.

The four modes of the serial port are referred to as modes 0, 1, 2 and 3. Mode 0 is the synchronous mode, and is commonly used to interface to shift registers for I/O expansion. In this mode the port outputs a pulse train on the TXD pin and either transmits or receives data on the RXD pin. Mode 1 is the standard asynchronous mode, 8 bits plus a stop and start bit are sent or received. Modes 2 and 3 handle 9 bits plus a stop and start bit. The difference between the two is, that in Mode 2 the serial port interrupt will not be activated unless the ninth data bit is a one; in Mode 3 the interrupt is activated whenever a byte is received. These two modes are commonly used for interprocessor communication.

Baud rates for all of the modes are controlled through the Baud Rate register. This is a byte wide register which is loaded sequentially with two bytes, and internally stores the value as a word. The least significant byte is loaded to the register followed by the most significant. The most significant bit of the baud value determines the clock source for the baud rate generator. If the bit is a one, the XTAL1 pin is used as the source, if it is a zero, the T2 CLK pin is used. The formulas shown in Figure 2-16 can be used to calculate the baud rates. The variable “B” is used to represent the least significant 15 bits of the value loaded into the baud rate register.

The baud rate register values for common baud rates are shown in Figure 2-17. These values can be used when XTAL1 is selected as the clock source for serial modes other than Mode 0. The percentage deviation from theoretical is listed to help assess the reliability of a given setup. In most cases a serial link will work if there is less than a 2.5% difference between the baud rates of the two systems. This is based on the assumption that 10 bits are transmitted per frame and the last bit of the frame must be valid for at least six-eights of the bit time. If the two systems deviate from theoretical by 1.25% in opposite directions the maximum tolerance of 2.5% will be reached. Therefore, caution must be used when the baud rate deviation approaches 1.25% from theoretical. Note that an XTAL1 frequency of 11.0592 MHz can be used with the table values for 11 MHz to provide baud rates that have 0.0 percent deviation from theoretical. In most applications, however, the accuracy available when using an 11 MHz input frequency is sufficient.

Serial port Mode 1 is the easiest mode to use as there is little to worry about except initialization and loading and unloading SBUF, the Serial port BUFfer. If parity is enabled, (i.e., PEN = 1), 7 bits plus even parity are used instead of 8 data bits. The parity calculation is done in hardware for even parity. Modes 2 and 3 are similar to Mode 1, except that the ninth bit needs to be controlled and read. It is also not possible to enable parity in Mode 2. When parity is enabled in Mode 3 the ninth bit becomes the parity bit. If parity is not enabled, (i.e., PEN = 0), the TB8 bit controls the state of the ninth transmitted bit. This bit must be set prior to each transmission. On reception, if PEN = 0, the RB8 bit indicates the state of the ninth received bit. If parity is enabled, (i.e., PEN = 1), the same bit is called RPE (Receive Parity Error), and is used to indicate a parity error.
The software used to communicate between processors is simplified by making use of Modes 2 and 3. In a basic protocol the ninth bit is called the address bit. If it is set high then the information in that byte is either the address of one of the processors on the link, or a command for all the processors. If the bit is a zero, the byte contains information for the processor or processors previously addressed. In standby mode all processors wait in Mode 2 for a byte with the address bit set. When they receive that byte, the software determines if the next message is for them. The processor that is to receive the message switches to Mode 3 and receives the information. Since this information is sent with the ninth bit set to zero, none of the processors set to Mode 2 will be interrupted. By using this scheme the overall CPU time required for the serial port is minimized.

A typical connection diagram for the multi-processor mode is shown in Figure 2-18. This type of communication can be used to connect peripherals to a desk top computer, the axis of a multi-axis machine, or any other group of microcontrollers jointly performing a task.
Figure 2-18. Multiprocessor Communication

Mode 0, the synchronous mode, is typically used for interfacing to shift registers for I/O expansion. The software to control this mode involves the REN (Receiver ENable) bit, the clearing of the RI bit, and writing to SBUF. To transmit to a shift register, REN is set to zero and SBUF is loaded with the information. The information will be sent and then the TI flag will be set. There are two ways to cause a reception to begin. The first is by causing a rising edge to occur on the REN bit, the second is by clearing RI with REN = 1. In either case, RI is set again when the received byte is available in SBUF.

2.3.5. A to D CONVERTER

Analog inputs are frequently required in a microcontroller application. The 8097 has a 10-bit A to D converter that can use any one of eight input channels. The conversions are done using the successive approximation method, and require 168 state times (42 microseconds with a 12 MHz clock.)

The results are guaranteed monotonic by design of the converter. This means that if the analog input voltage changes, even slightly, the digital value will either stay the same or change in the same direction as the analog input. When doing process control algorithms, it is frequently the changes in inputs that are required, not the absolute accuracy of the value. For this reason, even if the absolute accuracy of a 10-bit converter is the same as that of an 8-bit converter, the 10-bit monotonic converter is much more useful.

Since most of the analog inputs which are monitored by a microcontroller change very slowly relative to the 42 microsecond conversion time, it is acceptable to use a capacitive filter on each input instead of a sample and hold. The 8097 does not have an internal sample and hold, so it is necessary to ensure that the input signal does not change during the conversion time. The input to the A/D must be between ANGND and VREF. ANGND must be within a few millivolts of VSS and VREF must be within a few tenths of a volt of VCC.

Using the A to D converter on the 8097 can be a very low software overhead task because of the interrupt and HSO unit structure. The A to D can be started by the HSO unit at a preset time. When the conversion is complete it is possible to generate an interrupt. By using these features the A to D can be run under complete interrupt control. The A to D can also be directly
 controlled by software flags which are located in the AD_RESULT/AD_COMMAND Register, shown in Figure 2-19.

2.3.6. PWM REGISTER

Analog outputs are just as important as analog inputs when connecting to a piece of equipment. True digital to analog converters are difficult to make on a microprocessor because of all of the digital noise and the necessity of providing an on chip, relatively high current, rail to rail driver. They also take up a fair amount of silicon area which can be better used for other features. The A to D converter does use a D to A, but the currents involved are very small.

For many applications an analog output signal can be replaced by a Pulse Width Modulated (PWM) signal. This signal can be easily generated in hardware, and takes up much less silicon area than a true D to A. The signal is a variable duty cycle, fixed frequency waveform that can be integrated to provide an approximation to an analog output. The frequency is fixed at a period of 64 microseconds for a 12 MHz clock speed. Controlling the PWM simply requires writing the desired duty cycle value (an 8-bit value) to the PWM Register. Some typical output waveforms that can be generated are shown in Figure 2-20.

Converting the PWM signal to an analog signal varies in difficulty, depending upon the requirements of the system. Some systems, such as motors or switching power supplies actually require a PWM signal, not a true analog one. For many other cases it is necessary only to amplify the signal so that it switches rail-to-rail, and then filter it. Switching rail-to-rail means that the output of the amplifier will be a reference value when the input is a logical one, and the output will
be zero when the input is a logical zero. The filter can be a simple RC network or an active filter. If a large amount of current is needed a buffer is also required. For low output currents, (less than 100 microamps or so), the circuit shown in Figure 2-21 can be used.

The RC network determines how quiet the output is, but the quieter the output, the slower it can change. The design of high accuracy voltage followers and active filters is beyond the scope of this paper, however many books on the subject are available.

<table>
<thead>
<tr>
<th>DUTY CYCLE</th>
<th>PWM CONTROL REGISTER VALUE</th>
<th>OUTPUT WAVEFORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>00</td>
<td>HI LO</td>
</tr>
<tr>
<td>10%</td>
<td>25</td>
<td>HI LO</td>
</tr>
<tr>
<td>50%</td>
<td>128</td>
<td>HI LO</td>
</tr>
<tr>
<td>90%</td>
<td>230</td>
<td>HI LO</td>
</tr>
<tr>
<td>99.8%</td>
<td>255</td>
<td>HI LO</td>
</tr>
</tbody>
</table>

*This resistor limits Rise Time to reduce spikes and high frequency noise.

Figure 2-20. PWM Output Waveforms

Figure 2-21. PWM to Analog Conversion Circuity
3.0 BASIC SOFTWARE EXAMPLES

The examples in this section show how to use each I/O feature individually. Examples of using more than one feature at a time are described in section 4. All of the examples in this ap-note are set up to be used as listed. If run through ASM96 they will load and run on an SBE-96. In order to insure that the programs work, the stack pointer is initialized at the beginning of each program. If the programs are going to be used as modules of other programs, the stack pointer initialization should only be used at the beginning of the main program.

To avoid repetitive declarations the “include” file “DEMO96.INC”, shown in Listing 3-1, is used. ASM-96 will insert this file into the code file whenever the directive “INCLUDE DEMO96.INC” is used. The file contains the definitions for the SFRs and other variables. The include statement has been placed in all of the examples. It should be noted that some of the labels in this file are different from those in the file 8096.INC that is provided in the ASM-96 package.

3.1. Using the 8096's Processing Section

3.1.1. TABLE INTERPOLATION

A good way of increasing speed for many processing tasks is to use table lookup with interpolation. This can eliminate lengthy calculations in many algorithms. Frequently it is used in programs that generate sine waves, use exponents in calculations, or require some non-linear function of a given input variable. Table lookup can also be used without interpolation to determine the output state of I/O devices for a given state of a set of input devices. The procedure is also a good example of 8096 code as it uses many of the software features. Two ways of making a lookup table are described, one way uses more calculation time, the second way uses more table space.

Listing 3-1. Include File DEMO96.INC
In both methods the procedure is similar. Values of a function are stored in memory for specific input values. To compute the output function for an input that is not listed, a linear approximation is made based on the nearest inputs and nearest outputs. As an example, consider the table below.

If the input value was one of those listed then there would be no problem. Unfortunately the real world is never so kind. The input number will probably be 259 or something similar. If this is the case linear interpolation would provide a reasonable result. The formula is:

\[
\text{Delta Out} = \frac{\text{Upper Output} - \text{Lower Output}}{\text{Upper Input} - \text{Lower Input}} \times (\text{Actual Input} - \text{Lower Input})
\]

For the value of 259 the solution is:

\[
\begin{align*}
\text{Delta Out} &= \frac{900-400}{300-200} \times (259-200) \\
\text{Actual Output} &= 400 + 295 = 695
\end{align*}
\]

To make the algorithm easier, (and therefore faster), it is appropriate to limit the range and accuracy of the function to only what is needed. It is also advantageous to make the input step (Upper Input-Lower Input) equal to a power of 2. This allows the substitution of multiple right shifts for a divide operation, thus speeding up throughput. The 8096 allows multiple arithmetic right shifts with a single instruction providing a very fast divide if the divisor is a power of two.

For the purpose of an example, a program with a 12-bit output and an 8-bit input has been written. An input step of 16 (2**4) was selected. To cover the input range 17 words are needed, 255/16 + 1 word to handle values in the last 15 bytes of input range. Although only 12 bits are required for the output, the 16-bit architecture offers no penalty for using 16 instead of 12 bits.

The program for this example, shown in Listing 3-2, uses the definitions and equates from Listing 3-1, only the additional equates and definitions are shown in the code.

<table>
<thead>
<tr>
<th>Input Value</th>
<th>Relative Table Address</th>
<th>Table Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0001H</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>0002H</td>
<td>400</td>
</tr>
<tr>
<td>300</td>
<td>0003H</td>
<td>900</td>
</tr>
<tr>
<td>400</td>
<td>0004H</td>
<td>1600</td>
</tr>
</tbody>
</table>

```
$TITLE('INTER6.APT: Interpolation routine 1')
$INCLUDE(:Fl:DEM096.INC) ; Include demo definitions
RSEG at 22H
  IN VAL:  dab 1  ; Actual Input Value
  TABLE LOW: daw 1
  TABLE HIGH: daw 1
  IN DIF: daw 1  ; Upper Input - Lower Input
  IN DIFR equ IN_DIF ; byte
  TAB DIF: daw 1  ; Upper Output - Lower Output
  OUT: daw 1
  RESULT: daw 1
  OUT DIF: dsl 1  ; Delta Out
CSEG at 2080H
  LD SP, $100H
```

Listing 3-2. ASM-96 Code for Table Lookup Routine 1
If the function is known at the time of writing the software it is also possible to calculate in advance the change in the output function for a given change in the input. This method can save a divide and a few other instructions at the expense of doubling the size of the lookup table. There are many applications where time is critical and code space is overly abundant. In these cases the code in Listing 3-3 will work to the same specifications as the previous example.

Listing 3-2. ASM-96 Code for Table Lookup Routine 1 (Continued)

Listing 3-3. ASM-96 Code For Table Lookup Routine 2
By making use of the second lookup table, one word of RAM was saved and 16 state times. In most cases this time savings would not make much of a difference, but when pushing the processor to the limit, microseconds can make or break a design.

### 3.1.2. PL/M-96

Intel provides high level language support for most of its microprocessors and microcontrollers in the form of PL/M. Specifically, PL/M refers to a family of languages, each similar in syntax, but specialized for the device for which it generates code. The PL/M syntax is similar to PL/1, and is easy to learn. PLM-96 is the version of PL/M used for the 8096. It is very code efficient as it was written specifically for the MCS-96 family. PLM-96 most closely resembles PLM-86, although it has bit and I/O functions similar to PLM-51. One line of PL/M-code can take the place of many lines of assembly code. This is advantageous to the programmer, since code can usually be written at a set number of lines per hour, so the less lines of code that need to be written, the faster the task can be completed.

If the first example of interpolation is considered, the PLM-96 code would be written as shown in Listing 3-4. Note that version 1.0 of PLM-96 does not support 32-bit results of 16 by 16 multiplies, so the ASM-96 procedure "DMPY" is used. Procedure DMPY, shown in Listing 3-5, must be assembled and linked with the compiled PLM-96 program using RL-96, the relocator and linker. The command line to be used is:

```
RL96 PLMEX1.0OBJ, DMPY.OBJ, PLM96.LIB &
to PLMOUT.OBJ ROM (2080H-3FFFH)
```
Listing 3-4. PLM-96 Code For Table Lookup Routine 1

```assembly
/* PLM-96 CODE FOR TABLE LOOK-UP AND INTERPOLATION */

PLMEX:     DO;
DECLARE IN_VAL WORD PUBLIC;
DECLARE TABLE_LOW INTEGER PUBLIC;
DECLARE TABLE_HIGH INTEGER PUBLIC;
DECLARE TABLE_DIF INTEGER PUBLIC;
DECLARE RESULT INTEGER PUBLIC;
DECLARE OUT_DIF LONGINT PUBLIC;
DECLARE TEMP WORD PUBLIC;
DECLARE TABLE(17) INTEGER DATA {
  0000H, 2000H, 4000H, 6000H, 5000H, 6400H, 7200H, 7600H, 7B00H, 7D00H, 7600H, 6000H,
  5000H, 4B00H, 3400H, 2200H, 1000H};

DMFY:      PROCEDURE (A,B) LONGINT EXTERNAL;
DECLARE (A,B) INTEGER;
END DMYF;

LOOP:
   TEMP-SHR(IN_VAL,4); /* TEMP is the most significant 4 bits of IN_VAL */
   TABLE_LOW-TABLE(TMP); /* IF "TEMP" was replaced by "SHR(IN_VAL,4)" */
   TABLE_HIGH-TABLE(TMP+1); /* The code would work but the 8096 would */
   /* do two shifts */
   TABLE_DIF-TABLE.High-TABLE.Low;
   OUT_DIF=DMFY(TABLE_DIF,SIGNED(IN_VAL AND 0FH)) /16;
   OUT=SAR((TABLE_LOW+OUT_DIF),4); /* SAR performs an arithmetic right shift, */
   /* in this case 4 places are shifted */
   IF CARRY=0 THEN RESULT=OUT; /* Using the hardware flags must be done */
   ELSE RESULT=OUT+1; /* with care to ensure the flag is tested */
   /* in the desired instruction sequence */
   GOTO LOOP;
   /* END OF PLM-96 CODE */
END;
```

Listing 3-5. 32-Bit Result Multiply Procedure For PLM-96

```assembly
$TITLE('MULT.APT: 16*16 multiply procedure for PLM-96')

SP EQU 18H:word
CASEQ EXTRN PLMREG :long
CASEQ
PUBLIC DMYF ; Multiply two integers and return a
; longint result in AX, DX registers

DMFY: POP PLMREG+4 ; Load return address
POP PLMREG
MUL PLMREG,[SP]+ ; Load second operand and increment SP
BR [PLMREG+4]; Return to PLM code.

END

270061-22
```
Using PLM, code requires less lines, is much faster to write, and easier to maintain, but may take slightly longer to run. For this example, the assembly code generated by the PLM-96 compiler takes 56.75 microseconds to run instead of 30.75 microseconds. If PLM-96 performed the 32-bit result multiply instead of using the ASM-96 routine the PLM code would take 41.5 microseconds to run. The actual code listings are shown in Appendix A.

### 3.2. Using the I/O Section

#### 3.2.1. USING THE HSI UNIT

One of the most frequent uses of the HSI is to measure the time between events. This can be used for frequency determination in lab instruments, or speed/acceleration information when connected to pulse type encoders. The code in Listing 3-6 can be used to determine the high and low times of the signals on two lines. This code can be easily expanded to 4 lines and can also be modified to work as an interrupt routine.

Frequently it is also desired to keep track of the number of events which have occurred, as well as how often they are occurring. By using a software counter this feature can be added to the above code. This code depends on the software responding to the change in line state before the line changes again. If this cannot be guaranteed then it may be necessary to use 2 HSI lines for each incoming line. In this case one HSI line would look for falling edges while the other looks for rising edges. The code in Listing 3-7 includes both the counter feature and the edge detect feature.

The uses for this type of routine are almost endless. In instrumentation it can be used to determine frequency on input lines, or perhaps baud rate for a self adjusting serial port. Section 4.2 contains an example of making a software serial port using the HSI unit. Interfacing to some form of mechanically generated position information is a very frequent use of the HSI. The applications in this category include motor control, precise positioning (print heads, disk drives, etc.), engine control and

```asm
$TITLE('PULSE.APT: Measuring pulses using the HSI unit')
$INCLUDE(DM996.INC)

rsseg at 28H

HIGH_TIME:   dsw   1
LOW_TIME:    dsw   1
PERIOD:      dsw   1
HI_EDGE:     dsw   1
LO_EDGE:     dsw   1

rsseg at 2080H

LD     SP, $100H
LD 10C0, #00000001B    ; Enable HSI 0
LD 10C0, #00001111B    ; HSI 0 look for either edge

wait:   ADD  PERIOD, HIGH_TIME, LOW_TIME
     JBS  IOST1, 6, contin    ; If FIFO is full
     JBC  IOST1, 7, wait      ; Wait while no pulse is entered

contin: LD  AL, HSI_STATUS    ; Load status; Note that reading
        ; HSI_TIME clears HSI_STATUS
        LD  BX, HSI_TIME    ; Load the HSI_TIME
        JBS  AL, 1, hsi_hi  ; Jump if HSI.0 is high

hsi_lo: ST  BX, LOEDGE
        SUB  HIGH_TIME, LO_EDGE, HI_EDGE
        BR  wait

hsi_hi: ST  BX, HI_EDGE
        SUB  LOW_TIME, HI_EDGE, LO_EDGE
        BR  wait

END
```

Listing 3-6. Measuring Pulses Using The HSI Unit
transmission control. The HSI unit is used extensively in the example in section 4.3.

3.2.2. USING THE HSO UNIT

Although the HSO has many uses, the best example is that of a multiple PWM output. This program, shown in Listing 3-8, is simple enough to be easily understood, yet it shows how to use the HSO for a task which can be complex. In order for this program to operate, another program needs to set up the on and off time variables for each line. The program also requires that a HSO line not change so quickly that it changes twice between consecutive reads of I/O Status Register 0 (IOS0).

A very eye catching example can be made by having the program output waveforms that vary over time. The driver routine in Listing 3-10 can be linked to the above program to provide this function. Linking is accomplished using RL96, the relocatable linker for the 8096. Information for using RL96 can be found in the “MCS-96 Utilities Users Guide”, listed in the bibliography. In order for the program to link, the register dec-

```assembly
TITLE ('ENHSI.APT: ENHANCED HSI PULSE ROUTINE')
$INCLUDE(DEMO96.INC)
RSEG AT 28H
TIME: DSW 1
LAST_RISE: DSW 1
LAST_FALL: DSW 1
HSI_SO: DSB 1
IOS1_BAK: DSB 1
PERIOD: DSW 1
LOW_TIME: DSW 1
HIGH_TIME: DSW 1
COUNT: DSW 1

cseg at 2880H

init:
LD SP, @100H
LDB IOC1, #00100101B ; Disable HSO.4, HSO.5, HSI INT=first, ; Enable PWM,TXD,TIMER1_OVERFLOW_INT
LDB HSI_MODE, #1001100B
LDB IOC6, #00000111B

wait:
ANDB IOS1_BAK, #01111111B ; Clear IOS1_BAK.7
ORB IOS1_BAK, IOS1
JBC IOS1_BAK, #7,wait
ANDB HSI_SO, HSI_STATUS, #01010101B
LD TIME, HSI_TIME
JBS HSI_SO, #0,a_rise
JBS HSI_SO, #2,a_fall
BR no_cnt
a_rise: SUB LOW_TIME, TIME, LAST_FALL
SUB PERIOD, TIME, LAST_RISE
LD LAST_RISE, TIME
BR increment
a_fall: SUB HIGH_TIME, TIME, LAST_RISE
SUB PERIOD, TIME, LAST_FALL
LD LAST_FALL, TIME
increment:
INC COUNT
no_cnt: BR wait
END
```

Listing 3-7. Enhanced HSI Pulse Measurement Routine
TITLE ('HSOPWM.APT: 8096 EXAMPLE PROGRAM FOR PWM OUTPUTS')

; This program will provide 2 PWM outputs on HSO pins 0-2
; The input parameters passed to the program are:
; HSO_ON N: HSO on time for pin N
; HSO_OFF N: HSO off time for pin N

; Where: Times are in timer1 cycles
; N takes values from 0 to 3

;---------------------------------------------------------------

; INCLUDE(DEMO96.INC)

RSEG AT 28H

HSO_ON 0: DSW 1
HSO_OFF 0: DSW 1
HSO_ON 1: DSW 1
HSO_OFF 1: DSW 1
OLD_STAT: db 1
NEW_STAT: db 1

RSEG AT 2080H

LD SP $100H
LD HSO_ON 0, $100H
LD HSO_OFF 0, $400H
LD HSO_ON 1, $280H
LD HSO_OFF 1, $280H
ANDB OLD_STAT, I050, #0FH
XORB OLD_STAT, #0FH

wait: JBS IOS50, 6, wait
NOP

; For operation with interrupts 'store_stat;' would be the
; entry point of the routine
; Note that a DI or PUSHF might have to be added.

store_stat:

ANDB NEW_STAT, I050, #0FH
CMPB OLD_STAT, NEW_STAT
JE wait
XORB OLD_STAT, NEW_STAT

check_0:

JBC OLD_STAT, 0, check_1
JBS NEW_STAT, 0, set_off_0

set_on_0:

LDB HSO_COMMAND, #00010000B
ADD HSO_TIME, TIMER1, HSO_OFF_0
BR check_1

set_off_0:

LDB HSO_COMMAND, #00010000B
ADD HSO_TIME, TIMER1, HSO_ON_0

check_1:

JBC OLD_STAT, 1, check_done
JBS NEW_STAT, 1, set_off_1

set_on_1:

LDB HSO_COMMAND, #00010001B
ADD HSO_TIME, TIMER1, HSO_OFF_1
BR check_done

set_off_1:

LDB HSO_COMMAND, #00010001B
ADD HSO_TIME, TIMER1, HSO_ON_1

check_done:

LDB OLD_STAT, NEW_STAT

store current status and
wait for interrupt flag

BR wait
; use RET if 'wait' is called from another routine

END

Listing 3-8. Generating a PWM with the HSO
laration section (i.e., the section between “RSEG” and “CSEG”) in Listing 3-8 must be changed to that in Listing 3-9.

The driver routine simply changes the duty cycle of the waveform and sets the second HSO output to a frequency twice that of the first one. A slightly different driver routine could easily be the basis for a switching power supply or a variable frequency/variable voltage motor driver. The listing of the driver routine is shown in Listing 3-10.

Listing 3-9. Changes to Declarations for HSO Routine

Listing 3-10. Driver Module for HSO PWM Program
Since the 8096 needs to keep track of events which often repeat at set intervals it is convenient to be able to have Timer 2 act as a programmable modulo counter. There are several ways of doing this. The first is to program the HSO to reset Timer 2 when Timer 2 equals a set value. A software timer set to interrupt at Timer 2 equals zero could be used to reload the CAM. This software method takes up two locations in the CAM and does not synchronize Timer 2 to the external world.

To synchronize Timer 2 externally the T2 RST (Timer 2 ReSeT) pin can be used. In this way Timer 2 will get reset on each rising edge of T2 RST. If it is desired to have an interrupt generated and time recorded when Timer 2 gets reset, the signal for its reset can be taken from HSI.0 instead of T2RST. The HSI.0 pin has its own interrupt vector which functions independently of the HSI unit.

Another option available is to use the HSI.1 pin to clock Timer 2. By using this approach it is possible to use the HSI to measure the period of events on the input to Timer 2. If both of the HSI pins are used instead of the T2RST and T2CLK pins the HSIO unit can keep track of speed and position of the rotating device with very little software overhead. This type of setup is ideal for a system like the one shown in Figure 3-1, and similar to the one used in section 4.3.

In this system a sequence of events is required based on the position of the gear which represents any piece of rotating machinery. Timer 2 holds the count of the number of tooth edges passed since the index mark. By using HSI.1 as the input to Timer 2, instead of T2 CLK, it is possible to determine tooth count and time information through the HSI. From this information instantaneous velocity and acceleration can be calculated. Having the tooth edge count in Timer 2 means

---

**Figure 3-1. Using the HSIO to Monitor Rotating Machinery**
that the HSO unit can be used to initiate the desired tasks at the appropriate tooth count. The interrupt routine initiated by HSI.O can be used to perform any software task required every revolution. In this system, the overhead which would normally require extensive software has been done with the hardware on the 8096, thus making more software time available for control programs.

3.2.3. USING THE SERIAL PORT IN MODE 1

Mode 1 of the serial port supports the basic asynchronous 8-bit protocol and is used to interface to most CRTs and printers. The example in Listing 3-11 shows a simple routine which receives a character and then transmits the same character. The code is set up so that minor modifications could make it run on an interrupt basis. Note that it is necessary to set up some flags as initial conditions to get the routine to run properly. If it was desired to send 7 bits of data plus parity instead of 8 bits of data the PEN bit would be set to a one. Interprocessor communication, as described in section 2.3.4, can be set up by simply adding code to change RB8 and the port mode to the listing below. The hardware shown in Figure 3-2 can be used to convert the logic level output of the 8096 to ±12 or 15 volt levels to connect to a CRT. This circuit has been found to work with most RS-232 devices, although it does not conform to strict RS-232 specifications. If true RS-232 conformance is required then any standard RS-232 driver can be used.

```
$TITLE('SP.APT: SERIAL PORT DEMO PROGRAM')
$INCLUDE(DEMO96.INC)

cseg at 28H
CHR:  db 1
SPTMP: db 1
TEMPO: db 1
TEMPI: db 1
RCV_FLAG: db 1

cseg at 200CH
DCW sccr_port_int

cseg at 2080H
LD SP, $100H
LDB IOCI, #00100000B ; Set P2.0 to TXD
   ; Baud rate = input frequency / (64*baud_val)
   ; baud_val = (input frequency/64) / baud rate
baud_val equ 39 ; 39 = (12,000,000/64)/4800 baud
BAUD_HIGH equ, ((baud_val-1)/256) OR 80H ; Set MSB to 1
BAUD_LOW equ (baud_val-1) MOD 256

LDB BAUD_REG, #BAUD_LOW
LDB BAUD_REG, #BAUD_HIUGH
LDB SPCON, #01001001B ; Enable receiver, Mode 1
   ; The serial port is now initialized
STB SBURF, CHR
LDB TEMPO, #01000000B ; Set TI-temp
LDB INT_MASK, #01000000B ; Enable Serial Port Interrupt
EI
loop: BR loop ; Wait for serial port interrupt

ser_port_int:
PUSHF
rd again:
LDB SPTMP, SPSTAT ; This section of code can be replaced
   ; with "ORB TEMPO, SP STAT" when the
ORB TEMPO, SPTMP ; serial port TI and RI bugs are fixed
ANDB SPTMP, #01100000B
JNE rd again ; Repeat until TI and RI are properly cleared
```

Listing 3-11. Using the Serial Port in Mode 1
Listing 3-11. Using the Serial Port in Mode 1 (Continued)

```assembly
get_byte:
JBC TEMP0, 6, put_byte ; If RI-temp is not set
STB SBUF, CHR ; Store byte
ANDB TEMP0, #1011111B ; CLR RI-temp
LDB RCV_FLAG, #0FFH ; Set bit-received flag

put_byte:
JBC RCV_FLAG, 0, continue ; If receive flag is cleared
JBC TEMP0, 5, continue ; If TI was not set
LDB SBUF, CHR ; Send byte
ANDB TEMP0, #1101111B ; CLR TI-temp
ANDB CHR, #0111111B ; This section of code appends
CMPB CHR, #0DH ; an LF after a CR is sent
JNE clr_rcv
LDB CHR, #6AH ; continue
BR continue

clr_rcv:
CLRB RCV_FLAG ; Clear bit-received flag
continue:
POPF
RET
END
```

Figure 3-2. Serial Port Level Conversion
3.2.4. USING THE A TO D

The code in Listing 3-12 makes use of the software flags to implement a non-interrupt driven routine which scans A to D channels 0 through 3 and stores them as words in RAM. An interrupt driven routine is shown in section 4.1. When using the A to D it is important to always read the value using the byte read commands, and to give the converter 8 state times to start converting before reading the status bit.

Since there is no sample and hold on the A to D converter it may be desirable to use an RC filter on each input. A 100Ω resistor in series with a 0.22 pf capacitor to ground has been used successfully in the lab. This circuit gives a time constant of around 22 microseconds which should be long enough to get rid of most noise, without overly slowing the A to D response time.

4.0 ADVANCED SOFTWARE EXAMPLES

Using the 8096 for applications which consist only of the brief examples in the previous section does not really make use of its full capabilities. The following examples use some of the code blocks from the previous section to show how several I/O features can be used together to accomplish a practical task. Three examples will be shown. The first is simply a combination of several of the section 3 examples run under an interrupt system. Next, a software serial port using the HSIO unit is described. The concluding example is one of interfacing the HSI unit to an optical encoder to control a motor.

4.1. Simultaneous I/O Routines under Interrupt Control

A four channel analog to PWM converter can easily be made using the 8096. In the example in Listing 4 analog channels are read and 3 PWM waveforms are generated on the HSO lines and one on the PWM pin. Each analog channel is used to set the duty cycle of its associated output pin. The interrupt system keeps the whole program humming, providing time for a background task which is simply a 32 bit software counter. To show which routines are executing and in which

```
$TITLE('ATOD.APT: SCANNING THE A TO D CHANNELS')
$INCLUDE(DENO96.INC)
RSEG at 28H
  BL EQU BX:BYTE
  DL EQU DX:BYTE

RESULT_TABLE:
  RESULT_1: dw 1
  RESULT_2: dw 1
  RESULT_3: dw 1
  RESULT_4: dw 1

cseg at 2080H

start:  LD SP, $100H ; Set Stack Pointer
        CLR BX
next:   ADDB AD_COMMAND, BL, $100B ; Start conversion on channel indicated by BL register
        NOP
        NOP ; Wait for conversion to start
        check: JBS AD_RESULT_LO, 3, check ; Wait while A to D is busy
        LDB AL, AD_RESULT_LO ; Load low order result
        LDB AH, AD_RESULT_HI ; Load high order result
        ADDB DL, BL, BL ; DL-BL*2
        LODZ DL, DL
        ST AX, RESULT_TABLE[DX] ; Store result indexed by BL*2
        INCB BL ; Increment BL modulo 4
        ANDB BL, $03H
        BR next

END
```

Listing 3-12. Scanning the A to D Channels
order, Port 1 output pins are used to indicate the current status of each task. The actual code listing is included in Appendix B.

The initialization section, shown in Listing 4-1a, clears a few variables and then loads the first set of on and off times to the HSO unit. Note that 8 state times must be waited between consecutive loads of the HSO. If this is not done it is possible to overwrite the contents of the CAM holding register. An A/D interrupt is forced by setting the bit in the Interrupt Pending register. This causes the first A/D interrupt to occur just after the Interrupt Mask register is set and interrupts are enabled.

**Listing 4-1. Using Multiple I/O Devices**

```assembly
$TITLE ('8096 EXAMPLE PROGRAM FOR PWM OUTPUTS FROM A TO D INPUTS')
$PAGEWIDTH(130)
; This program will provide PWM outputs on HSO pins 0-2 and one on the PWN.
; The PWM values are determined by the input to the A/D converter.

$INCLUDE(DEMO96.INC)
RSEG AT 28H
DL EQU DX:BYTE

ON_TIME:
    PWM_TIME_1: DSW 1
    HSO_ON_0: DSW 1
    HSO_ON_1: DSW 1
    HSO_ON_2: DSW 1

RESULT_TABLE:
    RESULT_0: DSW 1
    RESULT_1: DSW 1
    RESULT_2: DSW 1
    RESULT_3: DSW 1

    NXT_ON_T: DSW 1
    NXT_OFF_0: DSW 1
    NXT_OFF_1: DSW 1
    NXT_OFF_2: DSW 1
    COUNT: DSI 1
    AD_NUM: DSW 1
    TMP: DSW 1
    HSO_PER: DSW 1
    LAST_LOAD: DSB 1

CSEG AT 2000H
    DCW start ; Timer_ovf_int
    DCW atod_done_int
    DCW start ; HSI_data_int
    DCW HSO_exec_int

CSEG AT 2080H
    start: LD SP, $100H ; Set Stack Pointer
            CLR AX
    wait:  DBC AX ; wait approx. 0.25 seconds for
            JNE wait ; SB to finish communications
            CLR B AD_NUM
            LD PWM_TIME_1, $080H
            LD HSO_PER, $100H
            LD HSO_ON_0, $040H
            LD HSO_ON_1, $008H
            LD HSO_ON_2, $000H
            ADD NXT_ON_T, Timer1, $100H
```

**Listing 4-1a. Initializing the A to D to PWM Program**
Listing 4-1a. Initializing the A to D to PWM program (Continued)

Listing 4-1b. Interrupt Driven HSO Routine
The HSO routine shown in Listing 4-1b is slightly different than the one in section 3. All of the HSO lines turn on at the same time, only the turn-off-time is varied between lines. This action is what is most commonly required for multiple PWM outputs and simplifies the software. A comparison is made between Timer 1 and the next HSO turn on time at the beginning of the routine. If the next turn on time has passed, then the on-times are loaded into the CAM, otherwise the off-times are loaded.

The maximum number of events in the CAM at any given time is 7. This occurs when the first line to turn off does so, causing the off-times for all of the lines to be loaded. For two of the lines there will be an off-time, an on-time, and the just loaded off-time. The other line (the one that just turned off) will have only the on-time and the just loaded off-time.

A/D conversions are performed by the code in Listing 4-1c about every 60 microseconds, 42 for the conversion, the rest for overhead. The A/D routine sets up the HSO and PWM on and off times. Since the A/D has a ten bit output, the most significant 8 bits are rounded up or down based on the least significant two bits.

### 4.2. Software Serial Port Using the HSIO Unit

There are many systems which require more than one serial port, an example is a system which must communicate with other computers and have an additional port for a local console. If the on-board UART is being used as an inter-processor link, the HSIO unit can be used to interface the 8096 to an additional asynchronous line. 

Figure 4-1 shows the format of a standard 10-bit asynchronous frame. The start bit is used to synchronize the receiver to the transmitter; at the leading edge of the START bit the receiver must set up its timing logic to sample the incoming line in the center of each bit. Following the start bit are the eight data bits which are transmitted least significant bit first. The STOP bit is set to the opposite state of the START bit to guar-
antee that the leading edge of the START bit will cause a transition on the line; it also provides for a dead time on the line so that the receiver can maintain its synchronization.

The remainder of this section will show how a full-duplex asynchronous port can be built from the HSIO unit. There are four sections to this code:

1. Interface routines. These routines provide a procedural interface between the interrupt-driven core of the software serial port and the remainder of the application software.

2. Initialization routine. This routine is called during the initialization of the overall system and sets up the various variables used by the software port.

3. Transmit ISR. This routine runs as an ISR (interrupt service routine) in response to an HSO interrupt interrupt. Its function is to serialize the data passed to it by the interface routines.

4. Receive ISRs. There are two ISRs involved in the receive process. One of them runs in response to an HSI interrupt and is used to synchronize the receive process at the leading edge of the start bit. The second receive ISR runs in response to an HSO generated software timer interrupt, this routine is scheduled to run at the center of each bit and is used to deserialize the incoming data.

The routines share the set of variables that are shown in Listing 4-2. These variables should be accessed only by the routines which make up the software serial port.

```c
; VARIABLES NEEDED BY THE SOFTWARE SERIAL PORT
; ---------------------------------------------
; rseg
rcve_state: db 1 ; indicates receive done
rxrdy equ 1 ; indicates receive overflow
rxoverrun equ 2 ; receive in progress flag
rcve_buf: db 1 ; used to double buffer receive data
sample_time: dw 1 ; records last receive sample time
serial_out: dw 1 ; Holds the output character+framing (start and stop bits) for transmit process.
baud_count: dw 1 ; Holds the period of one bit in units of T1 ticks.
txd_time: dw 1 ; Transition time of last Txd bit that was sent to the CAM
char: db 1 ; for test only

; COMMANDS ISSUED TO THE HSO UNIT
; ---------------------------------
; mark_command equ 0110101b ; timer1,set,interrupt on 5
; space_command equ 0010101b ; timer1,clr,interrupt on 5
; sample_command equ 0011000b ; software timer 0
; eject
```

Listing 4-2. Software Serial Port Declarations
The table also shows the declarations for the commands issued to the HSO unit. In this example HSI.2 is used for receive data and HSO.5 is used for transmit data, although other HSI and HSO lines could have been used.

The interface routines are shown in Listing 4-3. Data is passed to the port by pushing the eight-bit character into the stack and calling \texttt{char\_out}, which waits for any in-process transmission to complete and stores the character into the variable \texttt{serial\_out}. As the data is stored the START and STOP bits are added to the data bits. The routine \texttt{char\_in} is called when the application software requires a character from the port. The data is returned in the \texttt{ax} register in conformance to PLM 96 calling conventions. The routine \texttt{csts} can be called to determine if a character is available at the port before calling \texttt{char\_in}. (If no character is available \texttt{char\_in} will wait indefinitely).

The initialization routine is shown in Listing 4-4. This routine is called with the required baud rate in the

\begin{verbatim}
; char\_out: ; Output character to the software serial port
    pop    cx    ; the return address
    pop    bx    ; the character for output
    add    bx,bx ; to the character and leave as 16 bit
    ; wait for transmission to complete
    cmp    serial_out,0
    bne    wait_for_xmit
    ; put the formatted character in serial\_out
    ; return to caller
    csts:    ; Returns "true" (ax<>0) if char\_in has a character.
        clc    ax    ; Get a character from the software serial port
        inc    ax
    csts_exit:    ; return

    ; setup_serial_port:
    ; Called on system reset to initiate the software serial port.
    pop    cx    ; the return address
    push   bx    ; the baud rate (in decimal)
    mov    dx,\$0007h
    divu   ax,dx
    sar    ax,1
    st     ax,baud_count
    st     serial_out,0
    lod    serial_out,0
    st     rxrdy
    st     txrdy
    st     txd_time,0
    st     hso\_command,mark\_command
    st     hso\_time,txd_time
    clr    receive_variables
    clr    receive_reg
    ; call init\_receive ; setup to detect a start bit
    ; return
\end{verbatim}

\textbf{Listing 4-3. Software Serial Port Interface Routines}

\textbf{Listing 4-4. Software Serial Port Initialization Routine}
stack; it calculates the bit time from the baud rate and stores it in the variable `baud_count` in units of TIMER1 ticks. An HSO command is issued which will initiate the transmit process and then the remainder of the variables owned by the port are initialized. The routine `init_receive` is called to setup the HSI unit to look for the leading edge of the START bit.

The transmit process is shown in Listing 4-5. The HSO unit is used to generate an output command to the transmit pin once per bit time. If the `serial_out` register is zero a MARK (idle condition) is output. If the `serial_out` register contains data then the least significant bit is output and the register shifted right one place. The framing information (START and STOP bits) are appended to the actual data by the interface routines. Note that this routine will be executed once per bit time whether or not data is being transmitted. It would be possible to use this routine for additional low resolution timing functions with minimal overhead.

The receive process consists of an initialization routine and two interrupt service routines, `hsi_isr` and `software_timer_isr`. The listings of these routines are shown in Listings 4-6a, 4-6b, and 4-6c respectively. The

---

```assembly
; hso_isr: ; fields the hso interrupts and performs the serialization of the data.
; Note: this routine would be incorporated into the hso service strategy for an
; actual system.
cseg at 2006h
dcw hso_isr ; Set up vector

cseg
pushf
add txd_time,baud_count
cmp serial_out,0 ; if character is done send a mark
be send_mark
shr serial_out,1 ; else send bit 0 of serial_out and shift
bc send_mark ; serial_out left one place.

send_space:
1db hso_command,$space_command
1d hso_time,txd_time
br hso_isr_exit

send_mark:
1db hso_command,$mark_command
1d hso_time,txd_time

hso_isr_exit:
popf
ret
$eject
```

Listing 4-5. Software Serial Port Transmit Process

---

```assembly
; init_receive:
; called to prepare the serial input process to find the leading edge of
; a start bit.
1db loco,$00000000b ; disconnect change detector
1db hsi_mode,$00010000b ; negative edges on HSI.2
flush_fifo:
orb ios1_save,ios1
bnc ios1_save,7,flush_fifo_done
1d al,hsi_status
1d ar,hsi_time ; trash the fifo entry
andb ios1_save,$not($00h) ; clear bit 7.
br flush_fifo
flush_fifo_done:
1db loco,$00010000b ; connect HSI.2 to detector
ret
```

Listing 4-6a. Software Serial Port Receive Initialization

---

Listing 4-6. Receive Process
Listing 4-6b. Software Serial Port Start Bit Detect

Listing 4-6c. Software Serial Port Data Reception
start is detected by the *hsi_isr* which schedules a software timer interrupt in one-half of a bit time. This first sample is used to verify that the START bit has not ended prematurely (a protection against a noisy line). The software timer service routine uses the variable *rcve_state* to determine whether it should check for a valid START bit, deserialize data, or check for a valid STOP bit. When a complete character has been received it is moved to the receive buffer and *init_receive* is called to set up the receive process for the next character. This routine is also called when an error (e.g., invalid START bit) is detected.

Appendix C contains the complete listing of the routines and the simple loop which was used to initialize them and verify their operation. The test was run for several hours at 9600 baud with no apparent malfunction of the port.

### 4.3. Interfacing an Optical Encoder to the HSI Unit

Optical encoders are among one of the more popular devices used to determine position of rotating equipment. These devices output two pulse trains with edges that occur from 2 to 4000 times a revolution.

Frequently there is a third line which generates one pulse per revolution for indexing purposes. Figure 4-2 shows a six line encoder and typical waveforms. As can be seen, the two waveforms provide the ability to determine both position and direction. Since a microcontroller can perform real time calculations it is possible to determine velocity and acceleration from the position and time information.

Interfacing to the encoder can be an interesting problem, as it requires connecting mechanically generated electrical signals to the HSI unit. The problems arise because it is difficult to obtain the exact nature of the signals under all conditions.

The equipment used in the lab was a Pittman 9400 series gearmotor with a 600 line optical encoder from Vernitech. The encoder has to be carefully attached to the shaft to minimize any runout or endplay. Fortunately, Pitmann has started marketing their motors with ball bearings and optical encoders already installed. It is recommended that the encoder be mounted to the motor using the exact specifications of the encoder manufacturer and/or a good machine shop.

![Figure 4-2. Optical Encoder and Waveforms](image-url)
Digital filtering external to the 8096 is used on the encoder signals. The idealized signals coming from the encoder and after the digital filter are shown in Figure 4-3. The circuitry connecting the encoder to the 8096 requires only two chips. A one-shot constructed of XOR gates generates pulses on each edge of each signal. The pulses generated by Phase A are used to clock the signal from Phase B and vice versa. The hardware is shown in Figure 4-4. CMOS parts are used to reduce loading on the encoder so that buffers are not needed. Note that T2CLK is clocked on both edges of both filtered phases.

By using this method repetitive edges on a single phase without an edge on the other phase will not be passed on to the 8096. Repetitive edges on a phase can occur when the motor is stopped and vibrates or when it is changing direction. The digital filtering technique causes a little more delay in the signal at slow speeds than an analog filter would, but the simplicity trade off is worthwhile. The net effect of digital filtering is losing the ability to determine the first edge after a direction change. This does not affect the count since the first edge in both directions is lost.

If it is desired to determine when each edge occurs before filtering, the encoder outputs can be attached directly to the 8096. As these would be input signals, Port 0 is the most likely choice for connection. It would not be required to connect these lines to the HSI unit, as the information on them would only be needed when the motor is going very slowly.

The motor is driven using the PWM output pin for power control and a port pin for direction control. The 8096 drives a 7438 which drives 2 opto-isolators. These in turn drive two VFETs. A MOV (Metal Oxide Varistor, a type of transient absorber) is used to protect the VFETs, and a capacitor filters the PWM to get the best motor performance. Figure 4-5 shows the driver circuitry. To avoid noise getting into the 8096 system, the ±15 volt power supply is isolated from the 8096 logic power supply.

This is the extent of the external circuitry required for this example. All of the counting and direction detection are done by the 8096. There are two sections to the example: driving the motor and interfacing to the encoder. The motor driver uses proportional control with

![Figure 4-3. Filtered Encoder Waveforms](image)
some modifications and a braking algorithm. Since the main point of this example is I/O interfacing, the motor driver will be briefly described at the end of this section.

In order to interface to the encoder it is necessary to know the types of waveforms that can be expected. The motor was accelerated and decelerated many times using different maximum voltages. It was found that the

![Figure 4-4. Schematic of Optical Encoder to 8096 Interface](image1)

![Figure 4-5. Motor Driver Circuitry](image2)
motor would decelerate smoothly until the time between encoder edges was around 100 microseconds. At this point the motor would either continue to decelerate slowly, or would suddenly stop and reverse. The latter case is the one that was most problematic.

After a brief overview, each section of the program will be described separately, with the complete listing included in the Appendix D. In order to make debugging easier, as well as to provide insight into how the program is working, I/O port 1 is used to indicate the program status. This information consists of which routine the program is in and under which mode it is operating. The main program sections are: Main loop, HSI interrupt, Timer 2 check, and Motor drive. There are also minor sections such as initialization, timer overflow handling, and software timer handling. Tying everything together is some overhead and glue. Where the glue is not obvious it will be discussed, otherwise it can be derived from the listings.

The program is a main loop which does nothing except serve as a place for the program to go when none of the interrupt routines are being run. All of the processing is done on an interrupt basis.

There are three basic software modes which are invoked depending on the speed of the motor. The modes referred to as 0, 1 and 2, in order from slowest to fastest operation. When the program is running the operating mode is indicated by the lower 2 bits of Port 1, with the following coding:

<table>
<thead>
<tr>
<th>P1.0</th>
<th>P1.1</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>HSI looks at every edge</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>HSI looks at Phase A edges only</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>Timer 2 used instead of HSI</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>(alternate form of above)</td>
</tr>
</tbody>
</table>

The example is easiest to see if mode 2 is described first, followed by mode 1 then mode 0. In mode 2 Timer 2 is used to count edges on the incoming signal. A software timer routine, which is actually run using HSO.0, uses the Timer 2 value to update a LONG (32-bit) software counter labeled POSITION. The HSO routine runs every 260 microseconds. The HSO.0 interrupt is used instead of an actual software timer because of the ability to easily unmask it while other software timer routines are running.

In the code in Listing 4-7, the mode is first determined. For the first pass ignore the code starting with the label in_mode_1. Starting with in_mode_2 the counter is incremented or decremented based on bit zero of DIRECT. If DIRECT.O = 0 the motor is going backward, if it is 1 the motor is going forward. Next the count difference is checked to see if it is slow enough to go into mode 1. If not the routine returns to the code it was running when the interrupt occurred.

Listing 4-7. Motor Control HSO.0 Timer Routine

```assembly
CSEG AT 2280H

hs0_exec_int: ; Check mode - Update position in mode 2

PUSHF
ldb HSO_COMMAND,#30H
add HSO_TIME,TIMER1,HSO_dly
orb portl,#00100000B ; set P1.5
ld Timer_2,TIMER2
jbs Port1,In_mode2

in_model1: sub tmpl,Timer_2,old_t2 ; Check count difference in tmpl
cmp tmpl,#2
jnb end_swt0

set_mode0: jbc Port1,0,end_swt0 ; if already in mode 0
andb Port1,#11111100B ; Clear P1.0, P1.1 (set mode 0)
ldb last_stat,zero ; enable all HSI
br end_swt0

270061-50
```

6-43
If the pulse rate is slow enough to go to mode 1, the transition is made by enabling HSI.0 and HSI.1. Both of these lines are connected to the same encoder line, with HSI.0 looking for rising edges and HSI.1 looking for falling edges. The HSI_TIME register is read to speed up clearing the HSI FIFO and the LAST_TIME value is set up so the mode 1 routine does not immediately put the program into another mode. The HSI FIFO is then cleared, the Timer 2 value used throughout this routine is saved, and the routine returns.

This routine still runs in modes 0 and 1, but in an abbreviated form. The section of code starting with the label in_mode checks to see if the pulses are coming in so slowly that both HSI lines can be checked. If this is the case then all of the HSIs are enabled and the program returns. This routine is the secondary method for going from mode 1 to mode 0, the primary method is by checking the time between edges during the HSI routine, which will be described later.

The HSO routine will enable mode 0 after an edge is received. This could cause a problem if the last 2 edges on Phase A before the encoder stops were too close to enable mode 0. If this happened, mode 0 would not be enabled until after the encoder started again, resulting in missed edges on Phase B. Using the HSO routine to switch from mode 1 to mode 0 eliminates this problem.

Figure 4-6 shows a state diagram of how the mode switching is done. As can be seen, there are two sources for most of the mode decisions. This helps avoid problems such as the one mentioned above.

When either Mode 1 or Mode 0 is enabled the HSI interrupt routine performs the counting of edges, while the HSO routine only ensures that the correct mode is running. The routines for modes 0 and 1 share the same initialization and completion sections, with the main body of code being different.

The initialization routine is similar to many HSI routines. The flags are checked to ensure that the HSI FIFO data is valid, and then the FIFO is read. Next, the main body of code (for either mode 0 or mode 1) is...
NOTES:
Mode 0: HSI Examines edges on Phase A and B
Mode 1: HSI Examines edges on Phase A only
Mode 2: TIMER 2 stores edgecount

Figure 4-6. Mode State Diagram

Listing 4-8. Motor Control HSI Data Available Routine
run. At the end time and count values are saved and the holding register is checked for another event. Listing 4-8 contains the initialization and completion sections of the HSI routine.

Listing 4-9 is the main body of the Mode 1 routine. Before any calculations are done in Mode 1, the incoming pulse period is measured to see if it is too fast or too slow for mode 1. The time period between two edges is used so that the duty cycle of the waveform will not affect mode switching. If it is determined that Mode 2 should be set, Port 1.1 is set, all of the HSI lines are disabled, and the HSI fifo is cleared. If Mode 0 is to be set all of the HSI lines are enabled and the variable LAST__STAT is cleared. LAST__STAT = 0 is used as a flag to indicate the first HSI interrupt in Mode 0 after Mode 1. After the mode checking and setting are complete the incremental value in Timer 2 is used to update POSITION. The program then returns to the completion section of the routine.

There is a lot more code used in Mode 0 than in Mode 1, most of which is due to the multiple jump statements that determine the current and previous state of the HSI pins. In order to save execution time several blocks of code are repeated as can be seen in Listing 4-10. The first determination is that of which edge had occurred. If a Phase A edge was detected the LAST1__TIME and LAST2__TIME variables are updated so a reference to the pulse frequency will be available. These are the same variables used under Mode 1. A test is also made to see if the edges are coming fast enough to warrant being in Mode 1, if they are, the switch is made. If the last edge detected was on Phase B, the information is used only to determine direction.

```
In_mode_1: ; mode 1 HSI routine
    andb tmp1, hsl_e0, $01010000B
    jne no_cnt
    cmp_time:
        ld last2_time, last1_time
        ld last1_time, time
    cmp1:
        sub tmp1, time, last2_time
        jh check_max_time
    set_mode_2:
        orb Portl, $000000010B
        ldb IOC0, $00000000B
        move hsl, last1_bak
        andb hsl_bak, $0111111B
        orb hsl_bak, hsl
        jba hsl_bak, 7, mt_hsl
        br done_chk
    check_max_time:
        sub tmp1, time, last2_time
        cmp tmp1, max_hsl
        jnh done_chk
    set_mode_0:
        andb Portl, $11111110B
        ldb IOC0, $01010101B
        ldb last_stat, zero
    done_chk:
        sub delta_p, timer2, timer2_old
        jbc direct, 0, add_rev
    add_fwd:
        add position, delta_p
        addc position+2, zero
        br load_lasts
    add_rev:
        sub position, delta_p
        subc position+2, zero
        br load_lasts
$eject
```

Listing 4-9. Motor Control Mode 1 Routines
Listing 4-10. Motor Control Mode 0 Routines

In_mode_0:
    jbs    hsi_s0,0,a Rise
    jbs    hsi_s0,2,a Fall
    jbs    hsi_s0,4,b Rise
    jbs    hsi_s0,6,b Fall
    br    no_cnt

a_rise:    ld    last2_time,last1_time
            ld    last1_time,time
            cmp    time,min hsi
            jb    tst_stat

; set model -
= orb     Port1,$00000001B   ; Set Pl.0 (in mode 1)
= ldd     IOC0,$0000010B     ; Enable HSI 0 and 1

;tst_stat:
    jbs    last_stat,6,going fwd
    jbs    last_stat,4,going rev
    jbs    last_stat,2,change_dir
    cmpb    last_stat,zero
    je    first_time
    br    inp_err

a_fall:    ld    last2_time,last1_time
            ld    last1_time,time
            cmp    time,min hsi
            jb    tst_stat

; set model -
= orb     Port1,$00000001B   ; Set Pl.0 (in mode 1)
= ldd     IOC0,$0000010B     ; Enable HSI 0 and 1

;tst_stat:
    jbs    last_stat,4,going fwd
    jbs    last_stat,6,going rev
    jbs    last_stat,0,change_dir
    cmpb    last_stat,zero
    je    first_time
    br    inp_err

b_rise:    jbs    last_stat,0,going fwd
            jbs    last_stat,6,change_dir
            cmpb    last_stat,zero
            je    first_time
            br    inp_err

b_fall:    jbs    last_stat,2,going fwd
            jbs    last_stat,0,going rev
            jbs    last_stat,4,change_dir
            cmpb    last_stat,zero
            je    first_time
            br    inp_err

first_time:
    stb    hsi_s0,last_stat
    br    done_chk     ; add delta position

inp_err:    br    no_int

change_dir:
    nob    direct
    jbc    direct,0,going rev

going_fwd:
    orb     PORT2,$01000000B   ; set P2.6
    ldd     direct,$01
    addc    position,$01
    addc    position+2,zero
    br    st_stat

; going_rew:
    orb     PORT2,$1011111B    ; clear P2.6
    ldd     direct,$00
    subc    position,$01
    subc    position+2,zero
    br    st_stat

st_stat:    stb    hsi_s0,last_stat

270061-55
After mode correctness is confirmed and the \textit{LASTx\_TIME} values are updated the \textit{LAST\_STAT} (Last Status) variable is used to determine the current direction of travel. The \textit{POSITION} value is then updated in the direction specified by the last two edges and the status is stored. Note that the first time in Mode 0 after being in Mode 1, the Mode 1 \textit{done\_clk} routine is used to update \textit{POSITION}, instead of the routines \textit{going\_fwd} and \textit{going\_rev} from the Mode 0 section of code. The completion section of code is then executed.

Providing the PWM value to drive the motor is done by a routine running under Software Timer 1. The first section of code, shown in Listing 4-11a, has to do with calculating the position and timer errors. Listing 4-11b shows the next section of code where the power to be supplied to the motor is calculated. First the direction is checked and if the direction is reverse the absolute value of the error is taken. If the error is greater than 64K counts, the PWM routine is loaded with the maximum value. The next check is made to see if the motor is close enough to the desired location that the power to it should be reversed, (i.e., enter the Braking mode). If the motor is very close to the position or has slowed to the point that is likely to turn around, the \textit{Hold\_Position} mode is entered.

The determination of which modes are selected under what conditions was done empirically. All of the parameters used to determine the mode are kept in RAM so they can be easily changed on the fly instead of by re-assembling the program. The parameters in the listing have been selected to make the motor run, but have not been optimized for speed or stability. A diagram of the modes is shown in Figure 4-7.

In the \textit{Hold\_Position} mode power is eased onto the motor to lock it into position. Since the motor could be stopped in this mode, some integral control is needed, as proportional control alone does not work well when the error is small and the load is large. The \textit{BOOST} variable provides this integral control by increasing the output a fixed amount every time period in which the

\begin{verbatim}
Listing 4-11. Motor Control Software Timer 1 Routine

;------------------------------------------
; SOFTWARE TIMER ROUTINE 1
;------------------------------------------

CSEG AT 2600H

swt1_expired:

pushf
orb
db
add
id
sub
sub
ret

; set port1.7
;
;
; enable HSI, Tovf, HSO
;
;
; Calculate time & position error
;
;
; values are set
;
;
; Time Err = Desired time to finish - current time
;
;
; Pos Err = Desired position to finish - current position
;
;
; note that errors should get smaller so deltas will be
;
;
; positive for forward motion (time is always forward)

Listing 4-11a. Motor Control Software Position Counter

\end{verbatim}
chk_dir:
  cmp pos_err+2, zero
  jge go_forward

go_backward:
  neg
  ld br
  cmp pos_err, $00H
  jne ld_max
  br chk_brk

go_forward:
  ld br
  cmp pos_err+2, zero
  je chk_brk

ld_max:
  ld br
  cmp pos_err, pos_pnt
  jnb hold_position
  cmp pos_err, brake_point
  jnb ld_max
  br

braking:
  cmp pos_delta, zero
  jge
  neg pos_delta
  cmp pos_delta, vel_pnt
  jnb hold_position
  cmp pos_pnt, brake_point
  jnb ld_max
  br

hold_position:
  cmp pos_err, $02
  jnb cal_out
  clr boost
  br

output:
  mov tmp, max_hold
  mov tmp, pos_err
  mov tmp, zero
  jne no_boost
  add boost, $04
  add tmp, $2H
  br

no_boost:
  clr boost
  cmp tmp, max_hold
  jnb out put
  br

maxed:
  ld tmp, max_hold
  br

output:
  ld br

chk_sanity:

ld_pwr:
  ldb rpwr, pwm_pwr
  notb rpwr
  ldb pwm_dir, $2H

p2bkwd:
  dit
  andb port2, $01111111B
  ld br

p2fwd:
  dit
  orb port2, $10000000B

Listing 4-11b: Motor Control Power Algorithm
error does not get smaller. Once the error does get smaller, usually because the motor starts moving, BOOST is cleared.

A sanity check can be performed at this point to double check that the 8096 has proper control of the motor. In the example the worst that can happen is the proto-

```assembly
pwrset:
  cmp  time_err+2,zero ; do pos_table when err is negative
  jgt  end_p
  br   end_p

get_values:
  id   des_pos,[nxt_pos]+
  id   des_pos+2,[nxt_pos]+
  id   des_time+2,[nxt_pos]+
  id   max_pwr,[nxt_pos]+
  add  des_pos,offset
  addc des_pos+2,zero
  sub  last_pos_err,des_pos,position

end_p:  andb port1,#0111111B ; clear P1.7
  popf
  ret

pos_table:
  dcl   00000000H ; position 0
  dcl   0020H, 0080H ; next time, power
  dcl   00000000H ; position 1
  dcl   0040H, 0040H ; next time, power
  dcl   00000000H ; position 2
  dcl   0060H, 00c0H ; next time, power
  dcl   0fF0H, 8000H ; position 3
  dcl   0880H, 0080H ; next time, power
  dcl   00000000H ; position 4
  dcl   0580H, 0080H ; next time, power
  dcl   00000000H ; position 5
  dcl   00003000H ; position 6
  dcl   0070H, 00ffH ; next time, power
  dcl   00000000H ; position 7
  dcl   0090H, 0090H ; next time, power
  dcl   00000000H ; position 8
  dcl   00000000H ; next time, power

Listing 4.12. Motor Control Next Position Lookup
```
type will need to be reset, so the sanity check was not used. If one were desired, it could be as simple as checking a hardware generated direction indicator, or as complex as checking motor condition and other environmental factors.

After all checks have been made, the power value is loaded to the RPWR register using a software inversion to compensate for the hardware inversion. Direction is determined next and the power and direction are changed in adjacent instructions with interrupts disabled to prevent changing power without direction and vice versa.

To exercise the program logic the desired position is changed based on the time value using the code and lookup table shown in Listing 4-12.

The remaining sections of the program are relatively simple, but worth discussing briefly. The initialization routine initializes the I/O features and places several variables from ROM into RAM. Having these variables in RAM makes it easier to tweak the algorithm. Timer 1 is expanded into a 32-bit timer by the interrupt routine shown in Listing 4-13.

Software timer overhead is handled by the routine shown in Listing 4-14. In this routine the status of each timer bit is checked in a shadow register. If any of the timers have expired the appropriate routine is called.

```
CSEG AT 2200H
	timer_ovf_int:
		push
		orb
		chk_tl: jbc iosl_bak,5,tmr_int_done
		inc time+2
		andb iosl_bak,01011111B ; clear bit 5
		tmr_int_done:
		popf
		ret ; End of timer interrupt routine

Listing 4-13. Motor Control Timer Interrupt Routine

CSEG AT 2200H
	soft_tmr_int:
		pushf
		orb
		chk_swt0: jbc iosl_bak,6,chk_swt1
		andb iosl_bak,01111110B ; Clear bit 0 - end swt0
		chk_swt0+1: jbc iosl_bak,1,chk_swt2
		andb iosl_bak,01111110B ; Clear bit 1
		chk_swt1: jbc iosl_bak,2,chk_swt3
	
call swt1_expired
		chk_swt2: jbc iosl_bak,3,chk_swt4
	
call swt2_expired
		chk_swt3: jbc iosl_bak,4,swt_int_done
		andb iosl_bak,01111111B ; Clear bit 3
		call swt3_expired
		swt_int_done:
		popf
		ret ; End of software timer interrupt routine

Listing 4-14. Motor Control Software Timer Interrupt Handler

6-51
The last routine, shown in Listing 4-15, is the Software Timer 2 routine which outputs some variables to external RAM. It also keeps LAST1_Time within 1800H of Timer1 to prevent overflows from occurring when the Mode 0 and Mode 1 software check this variable.

A complete listing of the program as it is used in our lab can be found in Appendix D. For a given motor or encoder it will probably be necessary to change some of the time constants on the first page of the listing. With the motor used in our experimentation, pulses are missed from time to time when direction changes quickly. If the motor were not as fast to turn around or the encoder were mounted better these problems should disappear. The missing pulses occur when switching from Mode 1 to Mode 0; other than that no anomalies were found in the lab.

Prior to the version of code just discussed, several attempts were made, one of which could be used under certain constraints. It is possible to use only modes 2 and 0 to monitor the encoder, provided the encoder always operates smoothly and provides at least 200 microseconds between the last several edges of Phase A before reversing. This idea was originally tried because the motor was not characterized thoroughly at first, and caused problems because of the motor's tendency to stop suddenly when its speed was low.

If an encoder has a lower line count and therefore more time between output pulses the two mode solution can be used. The software for the two mode version can be easily extracted from the three mode version, so it will not be presented.

### 5.0 HARDWARE EXAMPLE

#### 5.1. EPROM Only Minimum System

The diagram in Figure 5-1 illustrates how to connect an 8096 in a minimum configuration system. Either 2764s or 27128s can be used in the system. Note that the lower EPROM contains the even bytes while the upper
one contains the odd bytes, and the addressing is not fully decoded. This means that the addressing on a 2764 will be such that the lower 4K of each EPROM is mapped at 0000H and 4000H while the upper 4K is mapped at 2000H. If the program being loaded is 16 Kbytes long the first half is loaded into the second half of the 2764s and vice versa. A similar situation exists when using 27128s.
This circuit will allow most of the software presented in this ap-note to be run. In a system designed for prototyping in the lab it may be desirable to buffer the I/O ports to reduce the risk of burning out the chip during experimentation. One may also want to enhance the system by providing RC filters on the A to D inputs, a precision VREF power supply, and additional RAM.

5.2. Port Reconstruction

If it is desired to fully emulate a 8396 then I/O ports 3 and 4 must be reconstructed. It is easiest to do this if the usage of the lines can be restricted to inputs or outputs on a port by port rather than line by line basis. The ports are reconstructed by using standard memory-mapped I/O techniques, (i.e., address decoders and latches), at the appropriate addresses. If no external RAM is being used in the system then the address decoding can be partial, resulting in less complex logic.

The reconstructed I/O ports will work with the same code as the on chip ports. The only difference will be the propagation delay in the external circuitry.
6.0 CONCLUSION

An overview of the MCS-96 family has been presented along with several simple examples and a few more complex ones. The source code for all of these programs are available in the Insite Users Library using order code AE-16. Additional information on the 8096 can be found in the Microcontroller Handbook and it is recommended that this book be in your possession before attempting any work with the MCS-96 family of products. Your local Intel sales office can assist you in getting more information on the 8096 and its hardware and software development tools.

7.0 BIBLIOGRAPHY

   Order number 122048-001.
   Order number 210918-002.
   Order number 122049-001.
   Order number 122134-001.
$TITLE('INTERI A96, Interpolation routine I')
$INCLUDE('FO.DEMO96 INC') ; Include demo definitions
$nullist ; Turn listing off for include file

IN_VAL: TABLE_LOW. TABLE_HIGH: Actual Input Value
IN_DIF: Upper Input - Lower Input
IN_DIFB: IN_DIF :byte
OUT: Delta Out
RESULT: Upper Output - Lower Output

LD SP, #100H
LD AL, IN_VAL ; Load temp with Actual Value
SHRB AL, #3 ; Divide the byte by 8
AND AL, #1FFFFFFFH ; Insure AL is a word address
This effectively divides AL by 2
so AL = IN_VAL/16

LD AX, AL ; Load byte AL to word AX
LD TABLE_LOW, TABLE [AX] ; TABLE_LOW is loaded with the value in the table at table location AX
IT

0

C

CD

0

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!

j

C:

CD

0

::!
$TITLE('INTER2 A96 Interpolation routine 2')

SOURCE STATEMENT
1      INCLUDE('FO_DEMO96.A96.INC') ; Include demo definitions
2      nolist ; Turn listing off for include file
3      END of include file

IN_VAL           dsb 1         ; Actual Input Value
TABLE_LOW:      dsw 1         ; Table value for function
TABLE_INC:      dsw 1         ; Incremental change in function
IN_DIF:         dsw 1         ; Upper Input - Lower Input
IN_DIFB:        equ IN_DIF  ; byte
OUT:            dsw 1         ; Delta Out
RESULT:         dsw 1

CSEG at 2080H
LD     SP, #100H        ; Initialize SP to top of reg. file
look:
LDB    AL, IN_VAL      ; Load temp with Actual Value
SHRB   AL, #3           ; Divide the byte by 8
ANDB   AL, #11111110B   ; Insure AL is a word address
This effectively divides AL by 2
so AL = IN_VAL/16
LDBIE  AX, AL          ; Load byte AL to word AX
LD     TABLE_LOW, VAL_TABLE[AX] ; TABLE_LOW is loaded with the value
in the value table at location AX
LD     TABLE_INC, INC_TABLE[AX] ; TABLE_INC is loaded with the value
in the increment table at location AX+2
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>209A 510F242A</td>
<td>ANDB IN_DIF, IN_VAL, #OFH</td>
<td>IN_DIF=least significant 4 bits of IN_VAL</td>
</tr>
<tr>
<td>209E AC2A2A</td>
<td>LDZIE IN_DIF, IN_DIFB</td>
<td>Load byte IN_DIFB to word IN_DIF</td>
</tr>
<tr>
<td>20A1 FE4C2B2A30</td>
<td>MUL OUT_DIF, IN_DIF, TABLE_INC</td>
<td>Output_difference = Input_difference*Incremental_change</td>
</tr>
<tr>
<td>20A6 4426302C</td>
<td>ADD OUT, OUT_DIF, TABLE_LOW</td>
<td>Add output difference to output generated with truncated IN_VAL as input</td>
</tr>
<tr>
<td>20AA 08042C</td>
<td>SHR OUT, #4</td>
<td>Round to 12-bit answer</td>
</tr>
<tr>
<td>20AD A4002C</td>
<td>ADDC OUT, zero</td>
<td>Round up if Carry = 1</td>
</tr>
<tr>
<td>20B0 C0E82C</td>
<td>no_inc: ST OUT, RESULT</td>
<td>Store OUT to RESULT</td>
</tr>
</tbody>
</table>
| 20B3 27CF | BR look | Branch to "look."

2100 cseg AT 2100H

2100 val_table:
2108 005006A0720078 DCW 0000H, 2000H, 3400H, 4COOH | A random function
2110 007B007D0076006D DCW 5000H, 6A00H, 7200H, 7800H
2118 005004B00340022 DCW 7B00H, 7600H, 7600H, 6D00H
2120 0010 DCW 5000H, 4800H, 3400H, 2200H
2122 inc_table:
2122 00024001B001100 DCW 0200H, 0140H, 0180H, 0110H | Table of incremental differences
2122 D000800600003D00 DCW 00D0H, 0080H, 0060H, 0030H |
2132 200090FF70FF00FF DCW 00020H, OFF90H, OFF70H, OFF00H |
213A 0EFE90FE0FE0F0E DCW 0FEEOH, 0FE90H, 0FE00H, 0FE00H |
2142 END

ASSEMBLY COMPLETED. NO ERROR(S) FOUND.
$TITLE('PLMEXi. PLM-96 Example Code for Table Lookup')

/* PLM-96 CODE FOR TABLE LOOK-UP AND INTERPOLATION */

1 PLMEX DO.

2 1 DECLARE IN_VAL WORD PUBLIC;
3 1 DECLARE TABLE_LOW INTEGER PUBLIC;
4 1 DECLARE TABLE_HIGH INTEGER PUBLIC;
5 1 DECLARE TABLE_DIF INTEGER PUBLIC;
6 1 DECLARE OUT INTEGER PUBLIC;
7 1 DECLARE RESULT INTEGER PUBLIC;
8 1 DECLARE OUT_DIF LONGINT PUBLIC;
9 1 DECLARE TEMP WORD PUBLIC;

10 1 DECLARE TABLE(17) INTEGER DATA (0000H, 2000H, 3400H, 4000H,
     6A00H, 7200H, 7800H, 7600H, 6D00H,
     5D00H, 4B00H, 3400H, 2200H, 7B00H, 7D00H, 7600H, 6D00H,
     5D00H, 6A00H, 7200H, 7800H, 7600H, 6D00H, 5D00H, 4B00H, 3400H, 2200H)

11 1 DMPY: PROCEDURE (A,B) LONGINT EXTERNAL;
12 2 DECLARE (A,B) INTEGER;
13 2 END DMPY;

14 1 LOOP TEMP=SHR(IN_VAL,41); /* TEMP is the most significant 4 bits of IN_VAL */
15 1 TABLE_LOW=TABLE(TEMP); /* If "TEMP" was replaced by "SHR(IN_VAL,41)" */
16 1 TABLE_HIGH=TABLE(TEMP+1); /* The code would work but the B096 would */
17 1 TABLE_DIF=TABLE_HIGH-TABLE_LOW;
18 1 OUT_DIF=DMPY(TABLE_DIF,SIGNED(IN_VAL AND OFH)) /16; 
19 1 OUT=SAR((TABLE_LOW+OUT_DIF),41); /* SAR performs an arithmetic right shift, */
   /* in this case 4 places are shifted */

270061-68
20 IF CARRY=0 THEN RESULT=OUT; /* Using the hardware flags must be done */
21 ELSE RESULT=OUT+1; /* with care to ensure the flag is tested */
22 END;
23 GOTO LOOP;
24 END;

/* END OF PLM-96 CODE */

---

### PL/M-96 Example Code for Table Lookup

#### ASSEMBLY LISTING OF OBJECT CODE

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0022A</td>
<td>A100001B</td>
<td>R LOOP: LD SP, #STACK</td>
</tr>
<tr>
<td>0026A</td>
<td>A00010</td>
<td>R LD TEMP, IN_VAL</td>
</tr>
<tr>
<td>0029B</td>
<td>080410</td>
<td>R SHR TEMP, #4H</td>
</tr>
<tr>
<td>002CB</td>
<td>4A1010C</td>
<td>R ADD TMPO, TEMP, TEMP</td>
</tr>
<tr>
<td>0030B</td>
<td>A31D000002</td>
<td>R LD TABLE_LOW, TABLE[TMPO]</td>
</tr>
<tr>
<td>0035B</td>
<td>A31D020004</td>
<td>R LD TABLE_HIGH, TABLE+2H[TMPO]</td>
</tr>
<tr>
<td>003AA</td>
<td>4B020406</td>
<td>R SUB TABLE_DIF, TABLE_HIGH, TABLE_LOW</td>
</tr>
<tr>
<td>003EA</td>
<td>CB06</td>
<td>R PUSH TABLE_DIF</td>
</tr>
<tr>
<td>0040A</td>
<td>410F0001C</td>
<td>R AND TMPO, IN_VAL, #OFH</td>
</tr>
<tr>
<td>0045B</td>
<td>C81C</td>
<td>R PUSH TMPO</td>
</tr>
<tr>
<td>0047B</td>
<td>EF0000</td>
<td>E CALL DMPY</td>
</tr>
<tr>
<td>004AB</td>
<td>OE041C</td>
<td>E SHRAL TMPO, #4H</td>
</tr>
<tr>
<td>004DB</td>
<td>A01E0E</td>
<td>R LD OUT_DIF+2H, TMP2</td>
</tr>
<tr>
<td>0050B</td>
<td>A01C0C</td>
<td>R LD OUT_DIF, TMPO</td>
</tr>
<tr>
<td>0053A</td>
<td>A00220</td>
<td>R LD TMP4, TABLE_LOW</td>
</tr>
<tr>
<td>0056B</td>
<td>0620</td>
<td>EXT TMP4</td>
</tr>
<tr>
<td>005CB</td>
<td>6A1E20</td>
<td>ADD TMPO, TMP4, TMP6, TMP2</td>
</tr>
<tr>
<td>005EB</td>
<td>OE0420</td>
<td>E SHRAL TMP4, #4H</td>
</tr>
<tr>
<td>0061B</td>
<td>A02008</td>
<td>R LD OUT, TMP4</td>
</tr>
<tr>
<td>0064B</td>
<td>B1FF1C</td>
<td>E LDB TMPO, #OFFH</td>
</tr>
<tr>
<td>0067B</td>
<td>D802</td>
<td>BC #0003</td>
</tr>
<tr>
<td>0069B</td>
<td>111C</td>
<td>CLRB TMPO</td>
</tr>
</tbody>
</table>

---

270061-69 270061-70
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0060</td>
<td>981C00</td>
<td>CMPB RO, TMPO</td>
</tr>
<tr>
<td>006E</td>
<td>D705</td>
<td>BNE @0001</td>
</tr>
<tr>
<td>0070</td>
<td>A0200A</td>
<td>STATEMENT 21</td>
</tr>
<tr>
<td>0073</td>
<td>2005</td>
<td></td>
</tr>
<tr>
<td>0075</td>
<td>@0001:</td>
<td></td>
</tr>
<tr>
<td>0075</td>
<td>A0080A</td>
<td></td>
</tr>
<tr>
<td>0078</td>
<td>070A</td>
<td></td>
</tr>
<tr>
<td>007A</td>
<td>0002:</td>
<td></td>
</tr>
<tr>
<td>007A</td>
<td>27AA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

**MODULE INFORMATION:**

- **CODE AREA SIZE**: 005AH 90D
- **CONSTANT AREA SIZE**: 0022H 34D
- **DATA AREA SIZE**: 0000H 0D
- **STATIC REGS AREA SIZE**: 0012H 18D

**PL/M-96 COMPILER**
- **PLMEXI**: PLM-96 Example Code for Table Lookup
- **ASSEMBLY LISTING OF OBJECT CODE**

**OVERLAYABLE REGS AREA SIZE**: 0000H 0D
**MAXIMUM STACK SIZE**: 0006H 6D

**PL/M-96 COMPILATION COMPLETE**: 0 WARNINGS, 0 ERRORS
MCS-96 MACRO ASSEMBLER MULT APT: 16*16 multiply procedure for PLM-96

SERIES-III MCS-96 MACRO ASSEMBLER, V1.0

SOURCE FILE: :F3: MULT APT
OBJECT FILE: :F3: MULT OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND: NOSB

ERR LOC OBJECT LINE SOURCE STATEMENT
1  $TITLE('MULT APT: 16*16 multiply procedure for PLM-96')
2
3
4  SP EQU 18H:word
5
6  rseg
7  EXTRN PLMREG :long
8
9  cseg
10
11  PUBLIC DMPY ; Multiply two integers and return a longint result in AX, DX registers
12
13
14  DMPY: POP PLMREG+4 ; Load return address
15
16
17  MUL PLMREG,[SP]+ ; Load second operand and increment SP
18
19  BR [PLMREG+4] ; Return to PLM code.

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.
SERIES-III MCS-96 RELOCATOR AND LINKER, V2.0
Copyright 1983 Intel Corporation

INPUT FILES: .F3:PLMEX1.OBJ, .F3:MULT.OBJ, PLM96 LIB
OUTPUT FILE: .F3:PLMOUT.OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND:
ROM(2080H-3FFFH)

INPUT MODULES INCLUDED:
·F3:PLMEX1.OBJ(PLMEX) 12/25/84
·F3:MULT.OBJ(MULT) 12/25/84
PLM96.LIB(PLMREG) 11/02/83

SEGMENT MAP FOR .F3:PLMOUT.OBJ(PLMEX):

<table>
<thead>
<tr>
<th>TYPE</th>
<th>BASE</th>
<th>LENGTH</th>
<th>ALIGNMENT</th>
<th>MODULE NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESERVED</strong></td>
<td>0000H</td>
<td>001AH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** GAP ***</td>
<td>001AH</td>
<td>0002H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REG</td>
<td>001CH</td>
<td>0008H</td>
<td>ABSOLUTE</td>
<td>PLMREG</td>
</tr>
<tr>
<td>REG</td>
<td>0024H</td>
<td>0012H</td>
<td>WORD</td>
<td>PLMEX</td>
</tr>
<tr>
<td>STACK</td>
<td>0036H</td>
<td>0006H</td>
<td>WORD</td>
<td></td>
</tr>
<tr>
<td>*** GAP ***</td>
<td>003CH</td>
<td>2044H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CODE</td>
<td>2080H</td>
<td>0003H</td>
<td>ABSOLUTE</td>
<td>PLMEX</td>
</tr>
<tr>
<td>CODE</td>
<td>2083H</td>
<td>0001H</td>
<td>WORD</td>
<td>PLMEX</td>
</tr>
<tr>
<td>CODE</td>
<td>2084H</td>
<td>007CH</td>
<td>WORD</td>
<td>PLMEX</td>
</tr>
<tr>
<td>CODE</td>
<td>2100H</td>
<td>000AH</td>
<td>BYTE</td>
<td>MULT</td>
</tr>
<tr>
<td>*** GAP ***</td>
<td>210AH</td>
<td>DEF6H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Symbol Table for `F3:PLMOUT.OBJ(PLMEX)`

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG WORD</td>
<td>0024H</td>
<td>IN_VAL</td>
</tr>
<tr>
<td>REG INTEGER</td>
<td>0026H</td>
<td>TABLE_LOW</td>
</tr>
<tr>
<td>REG INTEGER</td>
<td>0028H</td>
<td>TABLE_HIGH</td>
</tr>
<tr>
<td>REG INTEGER</td>
<td>002AH</td>
<td>TABLE_DIF</td>
</tr>
<tr>
<td>REG INTEGER</td>
<td>002CH</td>
<td>OUT</td>
</tr>
<tr>
<td>REG INTEGER</td>
<td>002EH</td>
<td>RESULT</td>
</tr>
<tr>
<td>REG LONGINT</td>
<td>0030H</td>
<td>OUT_DIF</td>
</tr>
<tr>
<td>REG WORD</td>
<td>0034H</td>
<td>TEMP</td>
</tr>
<tr>
<td>CODE ENTRY</td>
<td>2100H</td>
<td>DMPY</td>
</tr>
<tr>
<td>REG LONG</td>
<td>001CH</td>
<td>PLMREG</td>
</tr>
<tr>
<td>NULL NULL</td>
<td>003CH</td>
<td>MEMORY</td>
</tr>
<tr>
<td>NULL NULL</td>
<td>1FC4H</td>
<td>?MEMORY_SIZE</td>
</tr>
</tbody>
</table>

**Module:** PLMEX

**Module:** MULT

**Module:** PLMREG

**RL96 Completed:** 0 WARNING(S), 0 ERROR(S)

270061-74
SERIES-III MCS-96 MACRO ASSEMBLER, V1 0

SOURCE FILE F3 PULSE A96
OBJECT FILE F3 PULSE OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND NOSB

ERR LOC OBJECT
LINE SOURCE STATEMENT
1 $TITLE(‘PULSE A96 Measuring pulses using the HSI unit’)
2
3 $INCLUDE(DEM096 INC)
=1
4 $nolist : Turn listing off for include file
=1
5 End of include file

52
53
0028
54 rseg at 20H
55
56 HIGH_TIME. dsw 1
0028
57 LOW_TIME: dsw 1
002A
58 PERIOD: dsw 1
002C
59 HI_EDGE: dsw 1
0030
60 LO_EDGE: dsw 1
61
62
63
64 cseg at 2080H
65
66
67
68
69
70
71 wait: ADD PERIOD, HIGH_TIME, LOW_TIME
0080 A100011B
72 JBS IOSI. 6. contin ; If FIFO is full
0084 B10115
73 JBC IOSI. 7. wait ; Wait while no pulse is entered
0087 B10F03
74
75 contin: LDB AL, HSI_STATUS ; Load status; Note that reading
76 ; HSI_TIME clears HSI_STATUS
77
78 LDB BX, HSI_TIME ; Load the HSI_TIME
79
80 JBS AL. 1. hsi_hi ; Jump if HSI.0 is high
80A C03020
81 hsi_lo: ST BX, LO_EDGE
80C 482E3028
82 SUB HIGH_TIME, LO_EDGE, HI_EDGE
80E 27E4
83 BR wait
85
86
87 hsi_hi: ST BX, HI EDGE

2080
2084
2087
208A
208E
2091
2094
2097
209A
209D
20A0
20A4
20A6
270061-75
20A9 48302E2A
20AD 27DB
20AF

SUB LOW_TIME, HI_EDGE, LO_EDGE
BR wait
END

ASSEMBLY COMPLETED. NO ERROR(S) FOUND.

270061-76
SOURCE FILE  F3 ENHSI A96
OBJECT FILE  F3 ENHSI OBJ

ERR LOC  OBJECT
LINE  SOURCE STATEMENT
1  *TITLE ('ENHSI A96 ENHANCED HSI PULSE ROUTINE')
2
3  *INCLUDE(DEMO96 INC)
=1  4  *nolist, Turn listing off for include file
=1  52  , End of include file
0028  51  RSEG AT 20H
0028  53  36  TIME  DSW 1
002A  53  37  LAST_RISE.  DSW 1
002C  53  38  LAST_FALL.  DSW 1
002E  53  39  HSI_SO  DSB 1
002F  53  60  IOSI_BAK:  DSB 1
0030  53  61  PERIOD.  DSW 1
0032  53  62  LOW_TIME:  DSW 1
0034  53  63  HIGH_TIME:  DSW 1
0036  53  64  COUNT  DSW 1
2080  53  65  cseg at 2080H
2080 A100011B  55  init:  LD  SP, #100H
2080 B12516  56  70  LDB  IOCl, #00100101B ; Disable HSO. 4. HSO 5. HSI_INT=first.
2080 B19903  57  71  ; Enable PWM. TXD. TIMER1_OVFLOW_INT
2080 B10715  58  72  LDB  HSI_MODE, #10011001B ; set hsi. 1 - hsi. 0 +
2080 B17E2F  59  73  ; Enable hsi 0.1
2080 717FF7  60  74  LDB  IOCO, #00000011B ; T2 CLOCK=T2CLK. T2RST=T2RST
2080 515052E  61  75  ; Clear timer2
2080 850428  62  76  ; Clear other flags which may be needed
209A A0042B  63  77  ; If hsi is not triggered then
209A 500068  64  78  ; Jump to wait

SOURCE STATEMENT
STIFLE ('ENHSI A96 ENHANCED HSI PULSE ROUTINE')
SINCLUD0E(O96 INC)
$no list, Turn listing off for include file
End of include file
RSEG AT 20H
TIME  DSW 1
LAST_RISE.  DSW 1
LAST_FALL.  DSW 1
HSI_SO  DSB 1
IOSI_BAK:  DSB 1
PERIOD.  DSW 1
LOW_TIME:  DSW 1
HIGH_TIME:  DSW 1
COUNT  DSW 1

LDB  IOCl, #00100101B ; Disable HSO. 4. HSO 5. HSI_INT=first.
LDB  HSI_MODE, #10011001B ; set hsi. 1 - hsi. 0 +
LDB  IOCO, #00000011B ; T2 CLOCK=T2CLK. T2RST=T2RST

ANDB  IOSI_BAK, #01111111B ; Clear IOSI_BAK 7

ANDB  HSI_SO.HSI_STATUS, #01010101B
LD  TIME. HSI_TIME
A.5. Enhanced Pulse Measurement (Continued)

ASSEMBLY COMPLETED. NO ERROR(S) FOUND.
$TITLE('HSODRV.A96: Driver module for HSO PWM program')
HSODRV MODULE MAIN, STACKSIZE(8)

PUBLIC HSO_ON_0, HSO_OFF_0
PUBLIC HSO_ON_1, HSO_OFF_1
PUBLIC HSO_TIME, HSO_COMMAND
PUBLIC SP, TIMER1, IOS0

$INCLUDE(DEMO96. INC)
$nolist

rseg at 28H

0028 EXTRN OLD_STAT :byte
002A HSO_ON_0: dsw 1
002C HSO_OFF_0: dsw 1
002E HSO_ON_1: dsw 1
0030 HSO_OFF_1: dsw 1
0032 count: dsb 1

0200 cseg at 2080H

0208 strti: DI
0211 LD SP, #100H
0214 ANDB OLD_STAT, IOS0, #OFH
0217 XORB OLD_STAT, #OFH

028C initial:
028D LD CX, #0100H

loop: LD AX, #1000H
0294 SUB BX, AX, CX
0297 LD AX, CX

270061-79
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>209B C02B1C</td>
<td>88</td>
<td>ST AX, HSO_ON_0</td>
<td></td>
</tr>
<tr>
<td>209E C02A20</td>
<td>89</td>
<td>ST BX, HSO_OFF_0</td>
<td></td>
</tr>
<tr>
<td>20A1 0B011C</td>
<td>91</td>
<td>SHR AX, #1</td>
<td></td>
</tr>
<tr>
<td>20A4 0B0120</td>
<td>92</td>
<td>SHR BX, #1</td>
<td></td>
</tr>
<tr>
<td>20A7 C02C1C</td>
<td>93</td>
<td>ST AX, HSO_ON_1</td>
<td></td>
</tr>
<tr>
<td>20AA C02E20</td>
<td>94</td>
<td>ST BX, HSO_OFF_1</td>
<td></td>
</tr>
<tr>
<td>20AD EF0000</td>
<td>E 96</td>
<td>CALL wait</td>
<td></td>
</tr>
<tr>
<td>20B0 0722</td>
<td>98</td>
<td>INC CX</td>
<td></td>
</tr>
<tr>
<td>20B2 8900F22</td>
<td>99</td>
<td>CMP CX, #00F00H</td>
<td></td>
</tr>
<tr>
<td>20B4 D7D8</td>
<td>100</td>
<td>BNE loop</td>
<td></td>
</tr>
<tr>
<td>20B8 27D2</td>
<td>102</td>
<td>BR initial</td>
<td></td>
</tr>
<tr>
<td>20BA</td>
<td>104</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.
SERIES-III MCS-96 MACRO ASSEMBLER. V1.0

SOURCE FILE F3.HSOMOD A96
OBJECT FILE F3.HSOMOD OBJ

CONTROLS SPECIFIED IN INVOCATION COMMAND: NOSB

ERR LOC OBJECT

SOURCE STATEMENT

LINE
1 $TITLE('HSOMOD A96: 8096 PWM PROGRAM MODIFIED FOR DRIVER')
2 $PAGEWIDTH(130)
3 4 This program will provide 3 PWM outputs on HSO pins 0-2
5 6 The input parameters passed to the program are:
6 7 HSO_ON_N HSO on time for pin N
8 HSO_OFF_N HSO off time for pin N
9 10 Where: Times are in timer1 cycles
11 N takes values from 0 to 3
12 13
14 15
16 17 NOTE: Use this file to replace the declaration section of
17 18 the HSO PWM program from "$INCLUDE(DMO96.INC)" through
19 the line prior to the label "wait": Also change the last
20 branch in the program to a "RET".
21
22 0000
23 0001
24 0002
25 0003
26 0004
27 0005
28 0006
29 0007
30 0008
31 0009
32 0010
33 0011
34 0012
35 0013
36 0014
37 0015
38 0016
39 0017
40 0018
41 0019
42 0020
43 0021
44

SOURCE STATEMENT

LINE
1 $TITLE('HSOMOD A96: 8096 PWM PROGRAM MODIFIED FOR DRIVER')
2 $PAGEWIDTH(130)
3 4 This program will provide 3 PWM outputs on HSO pins 0-2
5 6 The input parameters passed to the program are:
6 7 HSO_ON_N HSO on time for pin N
8 HSO_OFF_N HSO off time for pin N
9 10 Where: Times are in timer1 cycles
11 N takes values from 0 to 3
12 13
14 15
16 17 NOTE: Use this file to replace the declaration section of
17 18 the HSO PWM program from "$INCLUDE(DMO96.INC)" through
19 the line prior to the label "wait": Also change the last
19 branch in the program to a "RET".
20
21 0000
22 0001
23 0002
24 0003
25 0004
26 0005
27 0006
28 0007
29 0008
30 0009
31 0010
32 0011
33 0012
34 0013
35 0014
36 0015
37 0016
38 0017
39 0018
40 0019
41 0020
42 0021
43 0022
44

This program will provide 3 PWM outputs on HSO pins 0-2. The input parameters passed to the program are:

- HSO_ON_N: HSO on time for pin N
- HSO_OFF_N: HSO off time for pin N

Where: Times are in timer1 cycles.
N takes values from 0 to 3.

NOTE: Use this file to replace the declaration section of the HSO PWM program from "$INCLUDE(DMO96.INC)" through the line prior to the label "wait". Also change the last branch in the program to a "RET".

```
D_STAT:       DSB 1
extrn HSO_ON_0:word, HSO_OFF_0:word
extrn HSO_ON_1:word, HSO_OFF_1:word
extrn HSO_TIME:word, HSO_COMMAND:byte
extrn TIMER1:word, IOSO:byte
extrn SP:word

public OLD_STAT
OLD_STAT:       dsb 1
NEW_STAT:       dsb 1

CSEG
PUBLIC wait
wait: JBS IOSO, 6. wait
NDP

For operation with interrupts 'store_stat:' would be the entry point of the routine.
Note that a DI or PUSHF might have to be added.
```
SERIES-III MCS-96 MACRO ASSEMBLER, VI 0

SOURCE FILE F3 SP A96
OBJECT FILE F3 SP OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND NOSB

ERR LOC OBJECT LINE SOURCE STATEMENT

1 2 $TITLE('SP A96: SERIAL PORT DEMO PROGRAM')
3 4 $INCLUDE(DEMO96 INC)
5 6 $nolist ; Turn listing off for include file
5 7 ; End of include file
5 8
5 9 0028
6 0rseg at 28H
6 56
6 02B
6 57 CHR: dsb 1
6 58 029
6 59 SPTemp: dsb 1
6 02A
6 60 TEMPO: dsb 1
6 02B
6 60 TEMPI: dsb 1
6 02C
6 61 RCV_FLAG: dsb 1
6 62 200C
6 63 cseg at 200CH
6 64 200C 9C20
6 65 DCW ser_port_int
6 66 2080
6 67 cseg at 2080H
6 68 2080 A1000118
6 69 LD SP, #100H
6 70 2084 B12016
6 71 LDB IOC1, #00010000B ; Set P2.0 to TXD
6 72 2084 B12016
6 73 BAUD_REG. IIBAUD_HIGH
6 74 BAUD_REG. IIBAUD_LOW
6 75
6 76 0027
6 77 baud_val equ 39 ; 39 = (12,000,000/64)/4800 baud
6 78 0080
6 79 BAUD_HIGH equ ((baud_val-1)/256) OR $00 ; Set MSB to 1
6 0026
6 80 BAUD_LOW equ (baud_val-1) MOD 256
6 81
6 82 2087 B1260E
6 83 LDB BAUD_REG, #BAUD_LOW
6 84 2089 B1800E
6 85 LDB BAUD_REG, #BAUD_HIGH

270061-83
loop: BR loop  ; Wait for serial port interrupt

209C 99  ser_port_int:  ; This section of code can be replaced with "ORB TEMPO. SP STAT" when the
209D 100  PUSHF  ; serial port TI and RI bugs are fixed
209D 101  rd_again  ; Set bit-received flag
209E 102  LDB  ; Set bit-received flag
209E 103  ORB  ; Set bit-received flag
209E 104  ANDB  ; Set bit-received flag
209E 105  JNE rd_again ; Set bit-received flag
209E 106
209E 107  get_byte.  ; Set bit-received flag
209E 108  JBC TEMPO. &. put_byte  ; If RI-temp is not set
209E 109  STB SBUF. CHR  ; Store byte
209E 110  ANDB TEMPO. #10111111B  ; Send byte
209E 111  LDB RCV_FLAG. #0FH  ; CLR TI-temp
209E 112
209E 113  put_byte  ; Send byte
209E 114  JBC RCV_FLAG. 0. continue  ; If receive flag is cleared
209E 115  JBC TEMPO. 5. continue  ; If TI was not set
209E 116  LDB SBUF. CHR  ; Send byte
209E 117  ANDB TEMPO. #10101111B  ; CLR TI-temp
209E 118
209E 119  ANDB CHR. #0111111B  ; This section of code appends
209E 120  CMPB CHR. #0DH  ; an LF after a CR is sent
209E 121  JNE clr_rcv  ; Send byte
209E 122  LDB CHR. #0AH  ; Send byte
209E 123  BR continue  ; Send byte
209E 124
209E 125  clr_rcv:  ; Send byte
209E 126  CLRB RCV_FLAG  ; Clear bit-received flag
209E 127  continue:  ; Clear bit-received flag
209E 128
209E 129  RET  ; CLEAR bit-received flag
209E 130  END  ; CLEAR bit-received flag

ASSEMBLY COMPLETED. NO ERROR(S) FOUND.
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0028</td>
<td></td>
<td>1</td>
<td>$TITLE('ATOD.A96: SCANNING THE A TO D CHANNELS')</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>$INCLUDE(DEMO96.INC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>$nolist; Turn listing off for include file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>End of include file</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>54</td>
<td>RSEG at 28H</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>56</td>
<td>BL EQU BX:BYTE</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>57</td>
<td>DL EQU DX:BYTE</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>59</td>
<td>RESULT_TABLE:</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>60</td>
<td>RESULT_1: dsw 1</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>61</td>
<td>RESULT_2: dsw 1</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>62</td>
<td>RESULT_3: dsw 1</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>63</td>
<td>RESULT_4: dsw 1</td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>2080</td>
<td></td>
<td>66</td>
<td>cseg at 2080H</td>
</tr>
<tr>
<td>2080</td>
<td></td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>2080</td>
<td></td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>2080</td>
<td></td>
<td>69</td>
<td>start: LD SP, #100H ; Set Stack Pointer</td>
</tr>
<tr>
<td>2084</td>
<td></td>
<td>70</td>
<td>CLR BX</td>
</tr>
<tr>
<td>2086</td>
<td></td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>2086</td>
<td></td>
<td>72</td>
<td>next: ADDB AD_COMMAND, BL, #10008 ; Start conversion on channel indicated by BL register</td>
</tr>
<tr>
<td>208A</td>
<td></td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>208A</td>
<td></td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>208B</td>
<td></td>
<td>75</td>
<td>NOP ; Wait for conversion to start</td>
</tr>
<tr>
<td>208B</td>
<td></td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>208C</td>
<td></td>
<td>77</td>
<td>check: JBS AD_RESULT_LO, 3 ; check ; Wait while A to D is busy</td>
</tr>
<tr>
<td>208F</td>
<td></td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>2092</td>
<td></td>
<td>79</td>
<td>LDB AL, AD_RESULT_LO ; Load low order result</td>
</tr>
<tr>
<td>2096</td>
<td></td>
<td>80</td>
<td>LDB AH, AD_RESULT_HI ; Load high order result</td>
</tr>
<tr>
<td>2099</td>
<td></td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>2099</td>
<td></td>
<td>82</td>
<td>ADDB DL, BL, BL ; DL=BL*2</td>
</tr>
<tr>
<td>209C</td>
<td></td>
<td>83</td>
<td>LDBZ DX, DL</td>
</tr>
<tr>
<td>209C</td>
<td></td>
<td>84</td>
<td>ST AX, RESULT_TABLE[DX] ; Store result indexed by BL*2</td>
</tr>
<tr>
<td>20A0</td>
<td></td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>20A0</td>
<td></td>
<td>86</td>
<td>INCB BL ; Increment BL modulo 4</td>
</tr>
</tbody>
</table>
SERIES-III MCS-96 MACRO ASSEMBLER, V1 0

SOURCE FILE: F3:A2DH5O.A96
OBJECT FILE: F3:A2DH50.OBJ

CONTROLS SPECIFIED IN INVOCATION COMMAND: NOSB

ERR LOC OBJECT  LINE  SOURCE STATEMENT
    1          $TITLE ('A2DH50.A96: GENERATING PWM OUTPUTS FROM A TO D INPUTS')
    2
    3 This program will provide 3 PWM outputs on HSO pins 0-2
    4 and one on the PWM.
    5
    6 The PWM values are determined by the input to the A/D converter.
    7
    8
    9
    10 $INClUS(DEX96 INC)
    11 $nolist / Turn listing off for include file
    =1 11
    =1 59 / End of include file

0028  61 RSEG AT 28H
0029  62
002A  63 DL  EQU DX:BYTE
002B  64
002C  65 ON_TIME:
002D  66 PWM_TIME_1:  DSW  1
002E  67 HSO_ON_0:    DSW  1
002F  68 HSO_ON_1:    DSW  1
0030  69 HSO_ON_2:    DSW  1
0031
0032  71 RESULT_TABLE:
0033  72 RESULT_0:    DSW  1
0034  73 RESULT_1:    DSW  1
0035  74 RESULT_2:    DSW  1
0036  75 RESULT_3:    DSW  1
0037
0038  76
0039  77 NXT_ON_T:    DSW  1
003A  78 NXT_OFF_0:   DSW  1
003B  79 NXT_OFF_1:   DSW  1
003C  80 NXT_OFF_2:   DSW  1
003D
003E  81 COUNT:       DSW  1
003F  82 AD_NUM:      DSW  1

270061-87
2000 87 cseg AT 2000H
2000 88 B020 DCW start ; Timer_ovf_int
2002 90 ID21 DCW Atod_done_int
2004 91 B020 DCW start ; HSI_data_int
2006 92 CC20 DCW HSO_exec_int
2080 94 *EJECT
2080 95 cseg AT 2080H
2080 98 A100011B start: LD SP, #100H ; Set Stack Pointer
2084 99 A01C CLR AX
2086 100 D01C DEC AX ; wait approx. 0.2 seconds for
2088 101 D7FC JNE wait ; SBE to finish communications
208A 1144 103 CLRB AD_NUM
208B 1144 104 A180002B LD PWM_TIME_1, #080H
2090 115 A100014B LD HSO_PER, #100H
2094 116 A140002A LD HSO_ON_0, #040H
2098 117 A180002C LD HSO_ON_1, #080H
209C 118 A1C0002E LD HSO_ON_2, #0CDH
20A0 121 450001A3B ADD NXT_ON_T, Timer!, #100H
20A5 122 B13606 LDB HSO_COMMAND, #00110110B ; Set HSO for timer1, set pin 0.1
20AB 123 A03804 LD HSO_TIME, NXT_ON_T ; with interrupt
20AC 124 FD NDB ; without interrupt
20AD 125 B12206 LDB HSO_COMMAND, #00100010B ; Set HSO for timer1, set pin 2
20B0 126 643804 ADD HSO_TIME, NXT_ON_T ; without interrupt
20B3 127 91074A ORB LAST_LOAD, #0000011B ; Last loaded value was set all pins
20B6 128 B10A08 LDB INT_MASK, #00001010B ; Enable HSO and A/D interrupts
20B9 129 B10A09 LDB INT_PENDING, #00001010B ; Fake an A/D and HSO interrupt
20BC 130 812206 loop: ORB Portl, #00000001B ; set P1.0
20C0 131 65010004 ADD COUNT, #01
20C4 132 A40042 ADDC COUNT+2, zero
20C7 133 71FEOF ANDB Portl, #11111110B ; clear P1.0
20CA 134 27F1 BR loop
20E8 136 *EJECT

270061-88
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181

20CC
20CC F2
20CD 91020F
2000 46380A46
2004 800046
2007 DE19
2009 644B3B
20DC B13605
200F A03804
20E2 FD
20E3 FD
20E4 B12206
20E7 A03804
20EA 91074A
20ED 802817
20F0 2026
20F2 304A23
20FF 442A3B3A
20F9 B11006
20FC A03A04
20FF FD
2100 442C3B3C
2104 B11106
2107 A03C04
210A FD
210B 442E3B3E
210F B11206
2112 A03E04
2115 71FB4A
2118 71FDD0F

HSO_exec_int
PUSHF
ORB Port1, #00000010B ; Set p1 1
SUB TMP.TIMER1, NXT_ON_T
CMP TMP.ZERO
JLT set_off_times
ADD NXT_ON_T, HSO_PER
LDB HSO_COMMAND, #00101110B ; Set HSO for timer1, set pin 0.1
NOP
LDB HSO_COMMAND, #00100010B ; Set HSO for timer1, set pin 2
NOP
LDB HSO_COMMAND, #00000010B ; Set HSO for timer1, set pin 3
NOP
NOP
LDB HSO_COMMAND, #00000011B ; Last loaded value was all ones
LDB PWM_CONTROL, PWM_TIME_1 ; Now is as good a time as any
BR check_done
JBC LAST_LOAD, 0. check_done
ADD NXT_OFF_0, NXT_ON_T, HSO_ON_0
LDB HSO_COMMAND, #00010000B ; Set HSO for timer1, clear pin 0
LD HSO_TIME, NXT_OFF_0
NOP
ADD NXT_OFF_1, NXT_ON_T, HSO_ON_1
LDB HSO_COMMAND, #00010001B ; Set HSO for timer1, clear pin 1
LD HSO_TIME, NXT_OFF_1
NOP
ADD NXT_OFF_2, NXT_ON_T, HSO_ON_2
LDB HSO_COMMAND, #00010010B ; Set HSO for timer1, clear pin 2
LD HSO_TIME, NXT_OFF_2
ANDB Port1, #11111000B ; Clear P1.1

check_done:
ANDB Port1, #1111101B ; Clear P1.1

270061-89
2118 F3  182  PDCPF
211C F0  183  RET
211D 184  #EJECT
211E 185 .........................................................
211F 186 .........................................................
2120 187 .........................................................
2121 188 .........................................................
2122 189 .........................................................
2123 190
2124 191 ATOD_done_int:
2125 192 PUSHF
2126 193 ORB  Port1. #00000100B  ; Set P1 2
2127 194
2128 195 ANDB  AL, AD_RESULT_LO.#11000000B  ; Load low order result
2129 196 LDB  AH, AD_RESULT_HI  ; Load high order result
212A 197 ADDB  DL, AD_NUM, AD_NUM  ; DL= AD_NUM +2
212B 198 LDBZE  DX, DL
212C 199 ST  AX, RESULT_TABLE[DX]  ; Store result indexed by DX
212D 200
212E 201 CMPB  AL, #01000000B
212F 202 JNH  no_rnd
2130 203 CMPB  AH, #OFFH  ; Don't increment if AH=OFFH
2131 204 JE  no_rnd
2132 205 INCB  AH
2133 206
2134 207 no_rnd: LDB  AL, AH  ; Align byte and change to word
2135 208 CLR8  AH
2136 209 ST  AX, ON_TIME[DX]
2137 210
2138 211 INCB  AD_NUM
2139 212 ANDB  AD_NUM, #03H  ; Keep AD_NUM between 0 and 3
213A 213
213B 214 next: ADDB  AD_COMMAND, AD_NUM, #1000B  ; Start conversion on channel
213C 215  indicated by AD_NUM register
213D 216 ANDB  Port1, #1111011B  ; Clear P1.2
213E 217 PDPF
213F 218 RET
2140 219
2141 220
2142 221 END

ASSEMBLY COMPLETED.  NO ERROR(S) FOUND.
SERIES-III MCS-96 MACRO ASSEMBLER, V1.0

SOURCE FILE: F3.SWPORT.A96
OBJECT FILE: F3.SWPORT.OBJ

CONTROLS SPECIFIED IN INVOCATION COMMAND: NOSB

ERR LOC OBJECT LINE
0000 63 iosl_save: dsb 1 ; Used to save contents of iosl
0001 64 rcev_state: dsb 1
0002 65 rrdy equ 1 ; Indicates receive done
0004 66 rsoverrun equ 2 ; Indicates receive overflow
0002 67 rip equ 4 ; Receive in progress flag
0002 68 rcev_buf: dsb 1 ; Used to double buffer receive data
0003 69 rcev_reg: dsb 1 ; Used to deserialize receive
0004 70 sample_time: dsw 1 ; Records last receive sample time
0006 71 serial_out: dsw 1 ; Holds the output character+framing (start and
0008 72 baud_count: dsw 1 ; Holds the period of one bit in
000a 73 trd_time: dsw 1 ; Transition time of last Txd bit that was
000c 74 char: dsb 1 ; For test only

commands issued to the HSO unit

mark_command equ 0110101b ; timer1.set.interrupt on 5
space_command equ 0010101b ; timer1.clr.interrupt on 5
sample_command equ 0011000b ; software timer 0

$end

270061-91
The 8096 starts executing here on reset, the program will initialize the software serial port and run a simple test to exercise it.

```
cseg at 2080h
reset_loc:
; The 8096 starts executing here on reset, the program will initialize the
; software serial port and run a simple test to exercise it.

0000h
0080h
0081h
0082h
0083h
0084h
0085h
0086h
0087h
0088h
0089h
008ah
1000h
1010h
1020h
1030h
1040h
1050h
1060h
1070h
1080h
1090h
10a0h
10b0h
10c0h
10d0h
10e0h
10f0h
2000h
2010h
2020h
2030h
2040h
2050h
2060h
2070h
2080h
2090h
20a0h
20b0h
20c0h
20d0h
20e0h
20f0h
3000h
3010h
3020h
3030h
3040h
3050h
3060h
3070h
3080h
3090h
30a0h
30b0h
30c0h
30d0h
30e0h
30f0h
4000h
4010h
4020h
4030h
4040h
4050h
4060h
4070h
4080h
4090h
40a0h
40b0h
40c0h
40d0h
40e0h
40f0h
5000h
5010h
5020h
5030h
5040h
5050h
5060h
5070h
5080h
5090h
50a0h
50b0h
50c0h
50d0h
50e0h
50f0h
6000h
6010h
6020h
6030h
6040h
6050h
6060h
6070h
6080h
6090h
60a0h
60b0h
60c0h
60d0h
60e0h
60f0h
7000h
7010h
7020h
7030h
7040h
7050h
7060h
7070h
7080h
7090h
70a0h
70b0h
70c0h
70d0h
70e0h
70f0h
8000h
8010h
8020h
8030h
8040h
8050h
8060h
8070h
8080h
8090h
80a0h
80b0h
80c0h
80d0h
80e0h
80f0h
9000h
9010h
9020h
9030h
9040h
9050h
9060h
9070h
9080h
9090h
90a0h
90b0h
90c0h
90d0h
90e0h
90f0h
a000h
a010h
a020h
a030h
a040h
a050h
a060h
a070h
a080h
a090h
a0a0h
a0b0h
a0c0h
a0d0h
a0e0h
a0f0h
b000h
b010h
b020h
b030h
b040h
b050h
b060h
b070h
b080h
b090h
b0a0h
b0b0h
b0c0h
b0d0h
b0e0h
b0f0h
c000h
c010h
c020h
c030h
c040h
c050h
c060h
c070h
c080h
c090h
c0a0h
c0b0h
c0c0h
c0d0h
c0e0h
c0f0h
```
0000 0000 0000 0000
0000 CC22
0002 CC20
0004 A10701E
0008 A120A1C
000C BC201C
000F C0081C
0012 C00600
0015 B16016
0018 3E15FD
001B 4414A0A
001F B13506
0022 A00A04
0025 1102
0027 1103
0029 1101
002B EF4800
002E E322
0030 0000
0032 CC22
0034 B10121
0037 6A2020
003A BB0006
003D D7FB
0040 C00620
0042 E322
0044 011C
0046 300102
0049 071C
004B 0100
004C F0

0000 132 cseg
133 setup_serial_port
135 Called on system reset to initiate the software serial port
138 pop cx the return address
139 pop bx the baud rate (in decimal)
140 ld ds,.#0007h ds:ax=.500.000 (assumes 12 Mhz crystal)
141 ld ax,.#A120h
142 div ax bx calculate the baud count (.500.000/baudrate)
143 st ax.baud_count
144 st 0.serial_out clear serial out
145 ldb loc1.101100000h Enable HSO S and Tsd
146 bbs ios0.6.% Wait for room in the HSO CAM
147 ; and issue a MARK command.
148 add tsd_time,timer1.20
149 ldb hso_command,MARK_command
150 ld hso_time,tsd_time
151 clrb rcve_buf clear out the receive variables
152 clrb rcve_reg
153 clrb rcve_state
154 call init_receive setup to detect a start bit
155 br [cx] return
156 execute
157 ;
158 char_out: Output character to the software serial port
159 ;
160 pop cx the return address
161 pop bx the character for output
162 ld (bx+1),#01h add the start and stop bits
163 ld bx.bx to the char and leave as 16 bit
164 add bx,bx
165 wait_for_xmit:
166 cmp serial_out.0 wait for serial_out=0 (it will be cleared by
167 bne wait_for_xmit the hso interrupt process)
168 st bx.serial_out put the formatted character in serial_out
169 br [cx] return to caller
170 ;
171 csts: Returns "true" (ax<>0) if char_in has a character.
172 ;
173 ;
174 clr ax
175 ;
176 ret
177 ;
178 char_in:
; Get a character from the software serial port
     ; wait for character ready
 004C 3001FD R 184   bbc rcve_state.0.char_in
 004F F2 R 185   pushf
 0050 71FE01 R 186   and b rcve_state.0.not(rxrdy)
 0053 AC021C R 187   ldbze al.rcve_buf
 0056 F3 R 188   popf
 0057 F0 R 189   ret
 190  ; leave the critical region
 191
 0058
 0058 hso_isr:
 192 ; Fields the hso interrupts and performs the serialization of the data
 193 ; Note: this routine would be incorporated into the hso service strategy
 194 ; for an actual system.
 196
 2006 197   cseg at 2006h
 2006 5800 R 198   dcw hso_isr
 199
 0058
 0058 F2
 0059 64080A R 200   pushf
 0059 64080A R 201   tdb add txd_time, baud_count
 0059 64080A R 202   cmp serial_out.0
 005C BB0006 R 203   ; if character is done send a mark
 005F DF0D R 204   be send_mark
 0061 080106 R 205   shr serial_out.0
 0064 B508 R 206   ; else send bit 0 of serial_out and shift
 0066
 0066 send_space:
 0066 B11506 R 207   ldb hso_command.0.space_command
 0066 A00A04 R 208   ldb hso_command.0.space_command
 006C 2006 R 209   ldb hso_time.txd_time
 006E 2006 R 210   ldb hso_isr_exit
 006E send_mark:
 006E B13506 R 211   ldb hso_command.0.mark_command
 0071 A00A04 R 212   ldb hso_command.0.mark_command
 0074 213   ldb hso_time.txd_time
 0074 214   ldb hso_time.txd_time
 0074 215   hso_isr_exit:
 0074 F3 R 216   popf
 0075 F0 R 217   ret
 0076 218  ; leave
 0076
 007b init_receive:
 199 ; Called to prepare the serial input process to find the leading edge of
 200 ; a start bit
 202 ;
 007b B10015 R 203   ldb ioc0.0noop0000000b ; disconnect change detector
 007b B12003 R 204   ldb hsi_mode.000100000b ; negative edges on HSI 2
 007c
 007c 901600 R 205   ldb hsi_mode.000100000b ; negative edges on HSI 2
 007c
 007c flush_fifo:
 207 ;
 007c 901600 R 208   orb iosl_save.iosl
 007F 370008 R 209   bbc iosl_save.7.flush_fifo_done
 0082 B0061C R 210   ldb al.hsi_status
 0085 A00A1C R 211   ldb al.hsi_time ; trash the fifo entry
008B 717F00  R  231    andb  iosl_save.#not(80h) ; clear bit 7.
008B 27EF    R  232    br  flush_fifo
008D         R  233    flush_fifo_done:
008D B1015   R  234    ldb  loc0.#00010000b ; connect HSI 2 to detector
0090 F0     R  235    ret
0090         R  236
0091         R  237
0092         R  238    hsi_isr:
0093         R  239    ; fields interrupts from the HSI unit, used to detect the leading edge
0094         R  240    of the START bit
0095         R  241    ; note: this routine would be incorporated into the HSI strategy of an actual
0096         R  242    system.
00A4         R  243
00A5         R  244
2004         R  245    cseg at 2004h
2004 9100    R  246    dcw  hsi_isr  ; setup the interrupt vector
2004         R  247
0091         R  248    cseg
0091 F2     R  249    pushf
0092 CB1C    R  250    push  ax
0094 B0061C  R  251    ldb  al.hsi_status
0097 A00404  R  252    ldi  sample_time.hsi_time
009A 341C15  R  253    bcc  al.4.exit_hsi
009D 3F15FD  R  254    bbs  iosl.7.7 ; wait for room in HSI holding reg
00AA A0081C  R  255    ldi  ax.baud_count ; send out sample command in 1/2
00A3 08011C  R  256    shr  ax.1 ; bit time
00A6 641C04  R  257    add  sample_time.ax
00A9 B18B06  R  258    ldb  hso_command.#sample_command
00AC C00404  R  259    st  sample_time.hso_time
00AF B10015  R  260    ldb  loc0.#00000000b ; disconnect hsi 2 from change detector
00B2         R  261    exit_hsi:
00B4 F3     R  262    pop  ax
00B5 F0     R  263    popf
00B6         R  264    ret
00B6         R  265    cseg
200A         R  266    cseg at 200ah
200A B600    R  267    cseg
009A         R  268    dcw  software_timer_isr  ; setup vector
009B         R  269
0096         R  270    cseg
0096 F2     R  271    pushf
0097 901600  R  272    orb  iosl_save.iosl
0098 71FE00  R  273    andb  iosl_save.#not(10h) ; clear bit 0
009D 51FC0100 R  274    andb  0.ceive_state.#ofch ; all bits except rrdy and overrun=0
009C D70C    R  275    bne  process_data
270061-95
process_start_bit:
  bbb  hsi_status.5. start_ok
  call  init_receive
  br   software_timer_exit

start_ok:
  orb  rcve_state.rip: set receive in progress flag
  br   schedule_sample

process_data:
  rcve_state.7. check_stopbit
  shrb rcve_reg.1
  hsi_status.5. datazero
  orb  rcve_reg.80H: set the new data bit
  datazero:
    addb rcve_state.10H: increment bit count
    br   schedule_sample

check_stopbit:
  bbb  hsi_status.5: DEBUG ONLY
  ldb  rcve_buf.rcve_reg
  orb  rcve_state.0rdy
  andb rcve_state.03H: Clear all but ready and overrun bits
  call  init_receive
  br   software_timer_exit

schedule_sample:
  sbb  ios0.7: wait for holding reg empty
  ldb  hso_command.0sample_command
  add  sample_time. baud_count
  st   sample_time. hso_time

software_timer_exit:
  popf
  ret
  end

ASSEMBLY COMPLETED. NO ERROR(S) FOUND.
$TITLE ('MOTCON.A96: Motor Control Example Program')

USE WITH C-STEP or later parts

December 20, 1984

$INCLUDE(DEMO96.INC)

Initial Values

min_hsi_t equ 30  ; min period for PHA edges in model 0 before mode 2
min_hsi_t equ 2*min_hsi_t  ; min period for PHA edges in model 0 before mode 1
max_hsi_t equ 3*min_hsi_t + min_hsi_t/2  ; max period for PHA edges in model 0

HSOO_dly_period equ 110 ; delay for HSO timer 0 (timed count of pulses)

s01_dly_period equ 250  ; delay for software timer 1
swt2_dly_period equ 250  ; delay for software timer 2
max_power equ 0ffh
max_brake equ 0ffh
maximum_hold equ 080h
brake_pnt equ 1200
position_pnt equ 100
velocity_pnt equ 16

RSEG at 024H

tmp:   dsl l
timer_2: dsl l
<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>002C</td>
<td>06</td>
<td>dsl1</td>
<td>tmr2_old</td>
</tr>
<tr>
<td>0030</td>
<td>07</td>
<td>dsl1</td>
<td>position</td>
</tr>
<tr>
<td>0034</td>
<td>08</td>
<td>dsl1</td>
<td>des_pos</td>
</tr>
<tr>
<td>003B</td>
<td>09</td>
<td>dsl1</td>
<td>pos_err</td>
</tr>
<tr>
<td>003C</td>
<td>0A</td>
<td>dsl1</td>
<td>delta_p</td>
</tr>
<tr>
<td>0040</td>
<td>0B</td>
<td>dsl1</td>
<td>time</td>
</tr>
<tr>
<td>0044</td>
<td>0C</td>
<td>dsl1</td>
<td>des_time</td>
</tr>
<tr>
<td>0048</td>
<td>0D</td>
<td>dsl1</td>
<td>time_err</td>
</tr>
<tr>
<td>004C</td>
<td>0E</td>
<td>dslw1</td>
<td>last_time_err</td>
</tr>
<tr>
<td>004E</td>
<td>0F</td>
<td>dslw1</td>
<td>last_pos_err</td>
</tr>
<tr>
<td>0050</td>
<td>10</td>
<td>dslw1</td>
<td>pos_delta</td>
</tr>
<tr>
<td>0052</td>
<td>11</td>
<td>dslw1</td>
<td>time_delta</td>
</tr>
<tr>
<td>0054</td>
<td>12</td>
<td>dslw1</td>
<td>last_pos</td>
</tr>
<tr>
<td>0056</td>
<td>13</td>
<td>dslw1</td>
<td>last1_time</td>
</tr>
<tr>
<td>0058</td>
<td>14</td>
<td>dslw1</td>
<td>last2_time</td>
</tr>
<tr>
<td>005A</td>
<td>15</td>
<td>dslw1</td>
<td>boost</td>
</tr>
<tr>
<td>005C</td>
<td>16</td>
<td>dslw1</td>
<td>tmp1</td>
</tr>
<tr>
<td>005E</td>
<td>17</td>
<td>dslw1</td>
<td>out_ptr</td>
</tr>
<tr>
<td>0060</td>
<td>18</td>
<td>dslw1</td>
<td>offset</td>
</tr>
<tr>
<td>0062</td>
<td>19</td>
<td>dslw1</td>
<td>nxt_pos</td>
</tr>
<tr>
<td>0064</td>
<td>1A</td>
<td>dslw1</td>
<td>Tpwr</td>
</tr>
<tr>
<td>0066</td>
<td>1B</td>
<td>dslw1</td>
<td>old_t2</td>
</tr>
<tr>
<td>0068</td>
<td>1C</td>
<td>dsl1</td>
<td>direct</td>
</tr>
<tr>
<td>0069</td>
<td>1D</td>
<td>dsl1</td>
<td>pwm_dir</td>
</tr>
<tr>
<td>006A</td>
<td>1E</td>
<td>dsl1</td>
<td>hsi_s0</td>
</tr>
<tr>
<td>006B</td>
<td>1F</td>
<td>dsl1</td>
<td>last_stat</td>
</tr>
<tr>
<td>006C</td>
<td>20</td>
<td>dsl1</td>
<td>pwm_pwr</td>
</tr>
<tr>
<td>006D</td>
<td>21</td>
<td>dsl1</td>
<td>ios1_bak</td>
</tr>
<tr>
<td>006E</td>
<td>22</td>
<td>dsl1</td>
<td>TR_COL</td>
</tr>
<tr>
<td>006F</td>
<td>23</td>
<td>dsl1</td>
<td>main_dly</td>
</tr>
<tr>
<td>0070</td>
<td>24</td>
<td>dsl1</td>
<td>max_pwr</td>
</tr>
<tr>
<td>0072</td>
<td>25</td>
<td>dsl1</td>
<td>max_brk</td>
</tr>
<tr>
<td>0074</td>
<td>26</td>
<td>dsl1</td>
<td>max_hold</td>
</tr>
<tr>
<td>0076</td>
<td>27</td>
<td>dsl1</td>
<td>vel_pnt</td>
</tr>
<tr>
<td>0078</td>
<td>28</td>
<td>dsl1</td>
<td>brk_pnt</td>
</tr>
<tr>
<td>007A</td>
<td>29</td>
<td>dsl1</td>
<td>pos_pnt</td>
</tr>
<tr>
<td>007C</td>
<td>2A</td>
<td>dsl1</td>
<td>HSO0_dly</td>
</tr>
<tr>
<td>007E</td>
<td>2B</td>
<td>dsl1</td>
<td>sw1_dly</td>
</tr>
<tr>
<td>0080</td>
<td>2C</td>
<td>dsl1</td>
<td>sw2_dly</td>
</tr>
<tr>
<td>0082</td>
<td>2D</td>
<td>dsl1</td>
<td>min_hsi</td>
</tr>
<tr>
<td>0084</td>
<td>2E</td>
<td>dsl1</td>
<td>min_hsi2</td>
</tr>
<tr>
<td>0086</td>
<td>2F</td>
<td>dsl1</td>
<td>max_hsi2</td>
</tr>
<tr>
<td>0100</td>
<td>30</td>
<td>dseg1</td>
<td>dseg at 100H</td>
</tr>
</tbody>
</table>

Note: Direct indicates the direction (1=forward, 0=reverse).
<table>
<thead>
<tr>
<th>PIN#</th>
<th>PORT</th>
<th>FLAG USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.0</td>
<td>timer_ovf_int</td>
<td></td>
</tr>
<tr>
<td>P1.1</td>
<td>atod_done_int</td>
<td></td>
</tr>
<tr>
<td>P1.2</td>
<td>hsi_data_int</td>
<td></td>
</tr>
<tr>
<td>P1.3</td>
<td>software timer 2 routine enter/leave</td>
<td></td>
</tr>
<tr>
<td>P1.4</td>
<td>Main program toggle</td>
<td></td>
</tr>
<tr>
<td>P1.5</td>
<td>software timer 0 routine enter/leave</td>
<td></td>
</tr>
<tr>
<td>P1.6</td>
<td>hsi_int enter/leave</td>
<td></td>
</tr>
<tr>
<td>P1.7</td>
<td>software timer 1 routine enter/leave</td>
<td></td>
</tr>
<tr>
<td>P2.6</td>
<td>Input direction (0=reverse, 1=forward)</td>
<td></td>
</tr>
<tr>
<td>P2.7</td>
<td>direction 0=rev, 1=fwd</td>
<td></td>
</tr>
</tbody>
</table>

```
cseg at 2000H
```
```
dcw timer_ovf_int
dcw atod_done_int
dcw hsi_data_int
dcw hso_execute_int
dcw hsi_int_0_int
dcw software timer_int
dcw ser_port_int
dcw external_int
```
```
cseg at 2080H
```
```
```
```
209D B12516 186 ldb IOC1, #00100101B ; Disable HSO 4, HSO 5, HSI_INT=first, 187 ; Enable PWM, TXD, TIMER1_0VRFLOW_INT 188
209E 01518 189 andb Port1, #1111110B ; clear P1.0.1 (set mode 0)
20A3 B19903 190 ldb HSI_mode, #00011001B ; set hsi.1.3 ~ hsi.0.2 + 191 ; Enable all hsi
20A6 B15715 192 ; T2 CLOCK=T2CLK, T2RST=T2RST 193 ; Clear timer2
209D B12516 194 \s| reject
209E 01518 195
20AE 0142 196 id zero_hsi_time
20BF 0140 197 clr time
20BF 0142 198 clr time+2
20C0 012B 199 clr timer_2
20C4 012A 200 clr timer_2+2
20C8 0130 201 clr position
20CE 0132 202 clr position+2
20D2 0154 203 clr last_pos
20D4 0134 204 clr des_pos
20D6 0136 205 clr des_pos+2
20D8 0144 206 clr des_time
20DC 0146 207 clr des_time+2
20E2 A00A56 208 id last1_time.Timer1
20E5 490008565B 209 sub last2_time.last1_time.#00H
20CA 116D 210 clr io1_st_bak
20CC 1109 211 clr int_pending
20CE A1F0015E 212 id out_ptr.#1FOH
20D2 A13000B2 213 id min_hsi.#min_hsi_t
20D6 A11000B4 214 id min_hsi.#min_hsi1_t
20DA A16900B6 215 id max_hsi.#max_hsi1_t
20DE A16E007C 216 id HSO0_dly.#HSO0_dly_period
20E2 A1FA007E 217 id swt1_dly.#swt1_dly_period
20E6 A1FA0080 218 id swt2_dly.#swt2_dly_period
20E9 A1FF0070 219 id max_power.#max_power
20EE A1FF0072 220 id max_brk.#max_brake
20F2 A1800074 221 id max_hold.#max_hold maximum
20F6 A180047B 222 id brk_pnt.#brake_pnt
20F9 A164007A 223 id pos_pnt.#position_pnt
20FD A1100716 224 id vel_pnt.#velocity_pnt
2102 A1002962 225 id nxt_pos.#pos_table
2106 B00006C 226 ldb pwm_pwr:zero
2109 B10169 227 ldb pwm_dir:#01h ; FORWARD
210C B12008 228
210F B13000 229 ldb int_mask,#00101101B ; Enable timer_ovf, hsi, swt. HSO.interrupts
2112 447COA04 230 ldb hso_command,#30H ; set HSO 0
2116 FD 232 hello
2117 FD 233 NQP
2118 B13906 234 ldb hso_command,#39H ; set swt 1
211B 447E0A04 235 add hso_time.Timer1.HSO0_dly
CSEG AT 2200H

timer_ovf_int

pushf

chck_t1. jbc ios1_bak.5. tmr_int_done

inc time2

andb ios1_bak.1111111B ; clear bit 5

tmr_int_done

popf

ret ; End of timer interrupt routine

CSEG AT 2220H

soft_tmr_int

pushf

chck_sw0. \n
jbc ios1_bak.0. chck_sw1

andb ios1_bak.11111108 ; Clear bit 0 - end swt0

chck_sw1

andb ios1_bak.1111010B ; Clear bit 1
call swt1_expired
jbc iosl_bak.2.chk_swt3
andb iosl_bak.#1111011B ; Clear bit 2
chk_swt3.

jbc iosl_bak.4.swt_int_done
andb iosl_bak.#1111011B ; Clear bit 3
call swt3_expired

swt_int_done:
popf ; END OF SOFTWARE TIMER INTERRUPT ROUTINE

#ject

; CSEG AT 2280H

CSEG AT 2280H

hso_exec_int.

; Check mode - Update position in mode 2

in_model1:
sub tmpl.Timer_2.old_t2 ; Check count difference in tmpl

in_model2:
sub delta_p.timer_2.tmr2_old ; get timer2 count difference

in_mode0:

set_mode0:
jbc Port1.0.end_swt0 ; if already in mode 0
andb Port1.#1111101B ; Clear P1.0. P1.1 (set mode 0)

set_mode2:
jbc Port1.0.end_swt0 ; if already in mode 0
andb Port1.#1111101B ; Clear P1.0. P1.1 (set mode 0)

(label used for 2839)

in_f: add position_delta_p
337  addc position+2, zero
338  br chk_mode
339
340  in_rev: sub position_delta_p
341  sub position+2, zero
342
343  chk_mode:
344  sub tmpl.Timer_2.old_t2 ; Check count difference in tmpl
345  cmp tmpl.#5 ; set model if count is too low
346  jgt end_swt0 ; count <= 5
347
348  set_model:
349  andb Port1.#1111111B ; Clear P1.1, set P1.0 (set mode 1)
350  orb Port1.#00000001B ; enable HSI 0 and 1
351
352  Id zero. HSI_TIME
353  sub last1_time.Timer1.min_hsi
354
355  *EXIT
356  
357  clr_hsi:
358  Id ZERO. HSI_TIME
359  andb iosl_bak.#0111111B ; clear bit 7
360  orb iosl_bak.iosl
361  jbs iosl_bak.7.clr_hsi ; If hsi is triggered then clear hsi
362
363  end_swt0:
364  Id old_t2.TIMER_2
365  andb port1.#11011111B ; clear P1.5
366  POPF
367  ret
368
369
370
371 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
372 ; ; SOFTWARE TIMER ROUTINE 2 ;
373 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
374
375  CSEG AT 2380H
376
377  swt2_expired:
378  pushf
379  Id hso_command.#3AH ; set swt_2
380  add hso_time.timer1.swt2_dly
381
382  orb port1.#00000100B ; set port 1.2
383  cmp out_ptr.#fffH
384  bnh pulsing
385  Id out_ptr.#f0H
This routine keeps track of the current time and position of the motor. The upper word of information is provided by the timer overflow routine.

This routine is used to save execution time for worst case loop.
The text appears to be a listing of assembly code. Here is a transcription of the code:

```
2421 3A6A2C 436    jbs  hsi_so.2.a_fall
2424 3C6A4D 437    jbs  hsi_so.4.b_rise
2427 3E6A5A 438    jbs  hsi_so.6.b_fall
242A 2094 439    br    no_cnt
242C A065B8 440    a_rise: ld    last2_time, last1_time
242E A04056 441    ld    last1_time, time
2432 6B3840 442    sub    time, last2_time
2435 B82420 443    cmp    time, min_hsi
2438 D906 444    jh    tst_stat
243A 91010F 445    : set model-
243D B10515 446    orb    Portl.#00000001B   ; Set P1.0 (in mode 1)
2440 447    tst_stat: Portl.#00000010B   ; Enable HSI 0 and 1
2444 448    ld    IOC0.#00000010B
2448 449    jbs  portl.#00000010B
244C DF46 450    jbs  last_stat.6.going_fwd
244E 27B2 451    jbs  last_stat.4.going_rev
2450 A05B58 452    jbs  last_stat.2.change_dir
2454 3E6B67 453    cmpb  last_stat.0
2458 3A6B50 454    je    first_time   ; first time in model0
245C 9B0068 455    br    no_int
2460 3B6B58 456    a_fall: ld    last2_time, last1_time
2464 A04056 457    ld    last1_time, time
2468 6B3840 458    sub    time, last2_time
246C B82420 459    cmp    time, min_hsi
2470 D906 460    jh    tst_stat
2474 91010F 461    : set model-
247D B10515 462    orb    Portl.#00000001B   ; Set P1.0 (in mode 1)
2480 463    tst_stat: Portl.#00000010B   ; Enable HSI 0 and 1
2484 464    ld    IOC0.#00000010B
2488 465    #EJECT   ; Enable HSIO and 1
248C 466    tst_stat:
2490 3C6B37 467    jbs  last_stat.6.going_fwd
2494 3E6B43 468    jbs  last_stat.6.going_rev
2498 3A6B5C 469    jbs  last_stat.0.change_dir
249C 9B006B 470    cmpb  last_stat.0
24A0 DF22 471    je    first_time   ; first time in model0
24A4 2057 472    br    no_int
24A8 3B6B27 473    b_rise: jbs  last_stat.0.going_fwd
24AC 3A6B33 474    jbs  last_stat.2.going_rev
24A0 3E6B1C 475    jbs  last_stat.6.change_dir
24B0 9B006B 476    cmpb  last_stat.0
24B4 DF12 477    je    first_time   ; first time in model0
24B8 2047 478    br    no_int
24C2 3A6B17 479    b_fall: jbs  last_stat.2.going_fwd
24C6 3B6B23 480    jbs  last_stat.0.going_rev
24CA 3E6B0C 481    jbs  last_stat.4.change_dir
24D0 9B006B 482    cmpb  last_stat.0
24D4 DF02 483    je    first_time   ; first time in model0
```

The code listing includes several assembly instructions and labels, indicating operations such as loading, comparing, branching, and conditional jumps. The labels, such as `a_rise`, `a_fall`, `b_rise`, `b_fall`, and `first_time`, suggest that the code is likely related to hardware control or some form of state transition in a system. The instructions are structured in a way that suggests they are part of a larger program, possibly for controlling some hardware device or system.
2492 2037 486 br no_int
487
2494 488 first_time
2494 C46B6A 489 stb hsi_s0.last_stat
2497 2072 490 br done_chk, add delta position
491
2499 492 change_dir:
2499 1268 493 notb direct
249B 30680F 494 no_inc: jbc direct.O.going_rev
495
249E 496 going_fwd:
249E 914010 497 orb PORT2.00000000B ; set P2 6
24A1 B1016B 498 ldb direct.001 ; direction = forward
24A4 65010030 499 add position.001
24AB A40032 500 addc position+2.zero
24AB 200D 501 br stat
24AD 502 going_rev:
24AD 71BF10 503 andb PORT2.01011111B ; clear P2 6
24BE B1016B 504 ldb direct.0100 ; direction = reverse
24BF 69010030 505 sub position.01
24BF A80032 506 subc position+2.zero
24BA 507 st_stat:
24BA C46B6A 509 stb hsi_s0.last_stat
24BD 511 load lasts
24BD A0282C 512 ldb tmr2.001 ; set timer 2
24C0 717F6D 513 no_cnt andb iosi_bak.00111111B ; clr bit 7
24C3 90156D 514 orb iosi_bak.001
24C6 376002 515 jbc iosi_bak.7.no_int
24C9 2746 516 again. br get_values
24CB 517 no_int andb port1.0011111B ; Clear P1 6
24CE F3 518 popf
24CF F0 519 ret ; end of hsi-data interrupt routine
24DD 520 $EJECT
24DE 521 #EJECT
24DF 522 
24DD 523 
24DD 524 In_mode_1 ; mode 1 HSI routine
24DD 525 
24DD 526 
24DD 527 andb tmp1hsi_s0.0011000B
24DD 528 jne no_cnt
24DD 529 cmp_time
24DD 530 last2_time, last1_time
24DD 531 last_time, time
24DD 532
24DD 533
24DD 534 cmp1. sub tmp1, time, last2_time
24DD 535 cmp tmp1.min_hsi

270061-A6
SOFTWARE TIMER ROUTINE 1

CSEG AT 2600H

swtl_expired:

CSEG AT 2600H
260E A0444A 586  ld  time_err+2.des_time+2 ; Calculate time & position error
2611 A0363A 587  ld  pos_err+2.des_pos+2
2614 4B404448 588  sub  time_err.des_time.time ; values are set
2618 AB424A 589  sub  time_err+2.time+2
261A AB304348 590  sub  pos_err.des_pos.position
261F AB323A 591  sub  pos_err+2.position+2
2622 FB 592
2623 4B4B4532 593  sub  time_delta.last_time_err.time_err
2627 A0444C 594  ld  last_time_err.time_err
262A AB304E50 595  sub  pos_delta.last_pos_err.pos_err
262E A0384E 596  ld  last_pos_err.pos_err
2631 88003A 597
2634 D60D 598
2636 033B 599
263A B10069 600
263E 89FFFE3A 601
2642 D70A 602  neg  pos_err  ; Pos_err = ABS VAL (pos_err)
2646 200D 603  cmp  pos_err+2.zero
2649 200D 604  jge  go_forward
2651 200D 605
2654 8003A 606  neg  pos_err  ; Pos_err = ABS VAL (pos_err)
2657 DF05 607  cmp  pos_err+2.zero
265A B0706C 608  je  chk_brk
2661 2031 609
2664 B10169 610  neg  pos_err  ; Pos_err = ABS VAL (pos_err)
2667 033B 611  cmp  pos_err+2.zero
266A 033B 612  jge  go_forward
266D B10069 613
2671 89FFFE3A 614
2677 D70A 615
267A 200D 616
2681 8003A 617
2684 DF05 618
2687 033B 619
268A 033B 620
268D B10069 621
2690 89FFFE3A 622
2696 D70A 623
2699 200D 624
269C B0706C 625
26A0 2031 626
26A3 B10169 627
26A6 8003A 628
26AF DF05 629
26B3 033B 630  neg  pos_err  ; Pos_err = ABS VAL (pos_err)
26B6 033B 631  cmp  pos_err+2.zero
26BA 033B 632  jge  go_forward
26C3 B10069 633
26C6 89FFFE3A 634
26CA D70A 635
26CB 200D 636
26CE B0706C 637
26D1 2031 638
26D4 B10169 639
26D7 8003A 640
26DA DF05 641
26DB 033B 642
26DE 033B 643
26E1 B10069 644
26E4 89FFFE3A 645
26E8 D70A 646
26EA 200D 647
26F3 B0706C 648
26F6 2031 649
26FA B10169 650  neg  pos_err  ; Pos_err = ABS VAL (pos_err)
26FE 89FFFE3A 651
270061-A8
2658 D9F1 634   jh  ld_max , position_error > brake_point
265A 635  braking:
265A 636  cmp  pos_delta, zero
265A 637  jge  chk_delta
265D 638  neg  pos_delta
265F 639  cmp  pos_delta, vel_pnt
2661 640  jnh  hold_position
2661 641  cmp  pos_delta, vel_pnt
2664 642  jnh  hold_position
2664 643  cmp  pos_delta, vel_pnt
2666 644  brake ldb  pwm_pwr, max_brk
2669 645  ldb  tmp_direct
266C 646  notb tmp
266C 647  ldb  pwm_dir, tmp
266E 648  br  ld_pwr
2671 649  2030 ..
2673 650  chck_position
2673 651  cmp  pos_err, #02
2677 652  jh  calc_out
2677 653  clr tmp+2
2677 654  jh  brake
2677 655  clr tmp+2
2677 656  br  output
267D 657  201F ..
267E 658  calc_out:
267F 659  mulub tmp, max_hold, #255
2683 660  mulu tmp, pos_err
2686 661  cmp  pos_delta, zero
268F 662  jne  no_bst
268F 663  add tmp, tmp+2.
2692 664  add tmp, boost, tmp+2
2694 665  br  ck_max
2694 666  no_bst clr boost
2696 667  ck max cmp tmp, max_hold
2699 668  jnh output
269B 669  maxed ldb tmp, max_hold
269E 670  output ldb pwm_pwr, tmp+2
26A1 671  2000 ..
26A1 672  stk
26A3 673  br  ld_pwr
26A3 674  ;
26A6 675  ;
26A6 676  $EJECT
26A9 677  ld_pwr
26A9 678  ldb rpw, pwm_pwr
26AB 680  ldb rpw
26AB 681  notb rpw
26AB 682  jbs pwm_dir,0,p2fwd
26AB 683
CSEG at 2800H

26AB 684 p2bkwd: DI
26AC 717F10 685 andb port2.01111111B ; clear P2.7
26AF B06417 686 ldb pwm_control.rpw
26B2 687 EI
26B3 200B 688 br puros
26B9 918010 689 p2fwd: DI
26B9 918010 690 orb port2.10000000B ; set P2 7
26BC FB 691 edx pwm_control.rpw
26C0 D225 692 EI
26C2 B9202962 693 puros:
26C8 A1002962 694 cmp time_err+2, zero ; do pos_table when err is negative
26C0 D225 695 jgt end_p
26C2 B9202962 696 br end_p
26C2 B9202962 697 ...
26C3 4B3034 698 cmp nxtpos.132pos_table)
26C4 0142 700 jlt get_vals ; jump if lower
26C8 A1002962 701 ld nxtpos.1pos_table
26CC 0142 702 clr time+2
26C0 D225 703 get_vals:
26C4 0142 704 ld despos.lnxtpos
26C4 0142 705 ld despos.1nxtpos
26C4 0142 706 ld des_time.1nxtpos
26C4 0142 707 ld max.pwr.1nxtpos
26C4 0142 708 ld max_brk.max.pwr
26C4 0142 709 ld max_brk.max.pwr
26C4 0142 710 add despos.offset
26C4 0142 711 addc despos.2, zero
26C4 0142 712 sub last_pos_err.despos.position
26C4 0142 713 end_p: andb port1.0011111B ; clear P1.7
26C4 0142 714 ...
26C4 0142 715 ...
26C4 0142 716 popf
26C4 0142 717 ret
26C4 0142 718 $EJECT
26C4 0142 719 ...
26C4 0142 720 ...
26C4 0142 721 main program ...
26C4 0142 722 ...
26C4 0142 723 ...
26C4 0142 724 ...
26C4 0142 725 ...
26C4 0142 726 ...
26C4 0142 727 ...
26C4 0142 728 MAIN_Prog:
26C4 0142 729 orb ios1_bak.ios1
26C4 0142 729 orb ios1_bak.ios1
26C4 0142 730 jbc ios1_bak.6.control
26C4 0142 731 andb ios1_bak.1011111B ; clear ios1_bak 6
26C4 0142 732 orb Port1.00010000B ; Compl Bit P1. 4
26C4 0142 733 call HSI_DATA_INT ; prevent lockup
ASSEMBLY COMPLETED. NO ERROR(S) FOUND.
An FFT Algorithm For MCS®-96 Products Including Supporting Routines and Examples

IRA HORDEN
ECO APPLICATIONS ENGINEER

© Intel Corporation, 1988

Order Number: 270189-002
1.0 INTRODUCTION

Intel's 8096 is a 16-bit microcontroller with processing power sufficient to perform many tasks which were previously done by microprocessors or special building block computers. A new field of applications is opened by having this much power available on a single chip controller.

The 8096 can be used to increase the performance of existing designs based on 8051s or similar 8-bit controllers. In addition, it can be used for Digital Signal Processing (DSP) applications, as well as matrix manipulations and other processing oriented tasks. One of the tasks that can be performed is the calculation of a Fast Fourier Transform (FFT). The algorithm used is similar to that in many DSP and matrix manipulation applications, so while it is directly applicable to a specific set of applications, it is indirectly applicable to many more.

FFTs are most often used in determining what frequencies are present in an analog signal. By providing a tool to identify specific waveforms by their frequency components, FFTs can be used to compare signals to one another or to set patterns. This type of procedure is used in speech detection and engine knock sensors. FFTs also have uses in vision systems where they identify objects by comparing their outlines, and in radar units to detect the doppler shift created by moving objects.

This application note discusses how FFTs can be calculated using Intel's MCS®-96 microcontrollers. A review of fourier analysis is presented, along with the specific code required for a 64 point real FFT. Throughout this application note, it is assumed that the reader has a working knowledge of the 8096. For those without this background the following two publications will be helpful:

1986 Microcontroller Handbook
Using the 8096, AP-248

These books are listed in the bibliography, along with other good sources of information on the MCS-96 product family and on Fast Fourier Transforms.

2.0 PROGRAM OVERVIEW

This application note contains program modules which are combined to create a program which performs an FFT on an analog signal sampled by the on-board ADC (Analog to Digital Converter) of the 8097. The results of the FFT are then provided over the serial channel to a printer or terminal which displays the results. In the applications listed in the previous section, the data from this FFT program would be used directly by another program instead of being plotted. However, the plotted results are used here to provide an example of what the FFT does. There are four program modules discussed in this application note:

FFTRUN - Runs a 64 point FFT on its data buffer. It produces 32 14-bit complex output values and 32 14-bit output magnitudes. A fast square root routine and log conversion routine are included.

A2DCON - Fills one of two buffers with analog values at a set sample rate. The sample time can be as fast as 50 microseconds using 8x9xBH components.

PLOTSP - Plots the contents of a buffer to a serially connected printer. Routines are provided for console output and hexadecimal to decimal conversion and printing.

FTMAIN - The main module which controls the other modules.

Each of the modules will be described separately. In order to better understand how the programs work together, a brief tutorial on FFTs will be presented first, followed by descriptions of the programs in the order listed above.

The final program uses 64 real data points, taken from either a table or analog input 1. Each of the data points is a 16-bit signed number. The processing takes 12.5 milliseconds when internal RAM is used as the data space. If external RAM is used, 14 milliseconds are required. Larger FFTs can be performed by slightly modifying the programs. A 256-point FFT would take approximately 65 milliseconds, and a 1024-point version would require about 300 milliseconds.

In the program presented, the analog sampling time is set for 1 sample every 100 microseconds, providing the 64 samples in 6.4 milliseconds. The sampling time can be reduced to around 60 microseconds per point by changing a variable, and less than 50 microseconds by using the 8x9xBH series of parts, since they have a 22 microsecond A to D conversion time.

The programs are set up to be run in a sequence instead of concurrently. This provides the fastest operation if the sampling speed were reduced to the minimum possible. For the fastest operation above about 80 microseconds a sample, the programs could be run concurrently, but this would require some minor modifications of the program. Figure 1 shows the timing of the program as presented.
These programs have run in the Intel Microcontroller Operation Application’s Lab and produced the results presented in this application note. Since the programs have not undergone any further testing, we cannot guarantee them to be bug proof. We, therefore, recommend that they be thoroughly tested before being used for other than demonstration purposes.

3.0 FOURIER TRANSFORMS

A Fourier Transform is a useful analytical tool that is frequently ignored due to its mathematically oriented derivations. This is unfortunate, since Fourier transforms can be used without fully understanding the mathematics behind them. Of course, if one understands the theory behind these transforms, they become much more powerful.

The majority of this application note deals with how a Fast Fourier Transform (FFT) can be used for spectrum analysis. This procedure takes an input signal and separates it into its frequency components. One can almost treat the FFT as a black box, which has as its output, the frequency components and magnitudes of the input signal, much like a spectrum analyzer.

From a mathematical standpoint, Fourier Transforms change information in the time domain into the frequency domain. The theory behind the Fourier transform stems from Fourier analysis, also called frequency analysis.

There are many books on the topic of Fourier analysis, several of which are listed in the bibliography. In this application note, only the pertinent formulas and uses will be presented, not their derivations.

The main idea in Fourier analysis is that a function can be expressed as a summation of sinusoidal functions of different frequencies, phase angles, and magnitudes. This idea is represented by the Fourier Integral:

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-j2\pi ft} \, dt$$

Where: $H(f)$ is a function of frequency

$$h(t)$$ is a function of time

Since

$$e^{-j\theta} = \cos \theta - j \sin \theta$$

$$H(f) = \int_{-\infty}^{\infty} h(t) (\cos (2\pi ft) - j \sin (2\pi ft)) \, dt$$

Figure 2 shows a rectangular pulse and its Fourier transform. Note that the results in the frequency domain are continuous rather than discrete. The horizontal axis in Figure 2a is frequency, while that of Figure 2b is time.

In a simplified case, the varying phase angles can be removed, and the integral changed to a summation, known as a Fourier Series. All periodic functions can be described in this way. This series, as shown below, can help provide a more graphical understanding of Fourier analysis.

$$y(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[ a_n \cos (2\pi nf_0 t) + b_n \sin (2\pi nf_0 t) \right]$$

for $n = 1$ to $\infty$

Where $f_0 = \frac{1}{T_0}$, the fundamental frequency.
This formula can also be represented in complex form as:

$$H(f) = \frac{\sin (2\pi T_0 f)}{2\pi T_0 f}$$

The Fourier series for a square wave is

$$A_n = \frac{\sin ((2k + 1) \cdot 2\pi f_0 t)}{(2k + 1)}$$

If these sinusoids are summed, a square wave will be formed. Figure 3 shows the graphical summation of the first 3 terms of the series. Since the higher frequencies contribute to the smoothness of the waveform at the corners, it is reasonable to compare only the flatness of the top of the waveform. The sharpness or risetime of the waveform can be determined by the highest frequency term being summed. With rise and fall times of 10% of the period, the waveform generated by the first 3 terms is within 20% of ideal. At 7 terms it is within 10%, and at 20 terms it is within 5%. With a 5% risetime, it is within 20% of ideal after 5 terms, 10% after 13 terms and 5% after 32 terms. Figure 4 shows the resultant waveforms after the summation of 7, 15 and 30 terms.

Fourier analysis can be used on equation 4 to find the coefficients $a_n$ and $b_n$. To make this process easier to use with a computer, a discrete form, rather than a continuous one, must be used. The discrete Fourier transform, shown in Equation 7, is a good approximation to the continuous version. The closeness of the approximation depends on several conditions which will be discussed later. The input to this transform is a set of N equally spaced samples of a waveform taken over a period of NT. The period NT is frequently referred to as the “Sampling Window”.

$$A_n = \sum_{n=0}^{\infty} a_n e^{2\pi j n t}$$

$$B_n = \sum_{n=0}^{\infty} b_n e^{2\pi j n t}$$

Figure 2. Rectangular Pulse and Its Fourier Transform

Figure 3. Graphical Summation of Sinewaves
Figure 4. Square Wave from Sinusoids
The value of the input at the discrete times \( t \), the Fourier Transform can be used to find the magnitude and phase shift of the signal at the frequencies \( f \).

A spectrum analyzer can provide similar information on an analog input signal by using analog filters to separate the frequency components. Regardless of its source, the information on component frequencies of a signal can be used to detect specific frequencies present in a signal or to compare one signal to another. Many lab experiments and product development tests can make use of this type of information. Using these methods, the purity of signals can be measured, specific harmonics can be detected in mechanical equipment, and noise bursts can be classified. All of this information can be obtained while still treating the FFT process as a black box.

Consider the discrete transform of a square wave as shown in Figure 5. Note that the component magnitudes, as shown in the series of Equation 6, are shown in a mirrored form in the transform. This will happen whenever only real data is used as the FFT input, if both real and imaginary data were used the output would not be guaranteed to be symmetrical. For this reason, there is duplicate information in the transform for many applications. Later in this section a method to make the most of this characteristic is discussed.

### Figure 5. Discrete Transform of a Square Wave
If one looks at Equation 8, it can be seen that the calculation of a discrete Fourier transform requires \(N^2\) complex multiplications. If \(N\) is large, the calculation time can easily become unrealistic for real-time applications. For example, if a complex multiplication takes 40 microseconds, at \(N = 16\), 10 milliseconds would be used for calculation, while at \(N = 128\), over half a second would be needed. A Fast Fourier Transform is an algorithm which uses less multiplications, and is therefore faster. To calculate the actual time savings, it is first necessary to understand how a FFT works.

### 4.0 THE FFT ALGORITHM

The FFT algorithm makes use of the periodic nature of waveforms and some matrix algebra tricks to reduce the number of calculations needed for a transform. A more complete discussion of this is in Appendix A, however, the areas that need to be understood to follow the algorithm are presented here. This information need not be read if the reader's intent is to use the program and not to understand the mathematical process of the algorithm.

To simplify notation the following substitutions are made in Equation 8.

\[
W = e^{-j2\pi/N}
\]

\[
k = kT
\]

\[
n = \frac{n}{NT}
\]

The resultant equation being

\[
x(n) = \sum_{k=0}^{N-1} n(k)W^{nk}
\] (9)

Expressed as a matrix operation

\[
\begin{bmatrix}
X(1) \\
X(2) \\
X(3) \\
X(N-1)
\end{bmatrix} =
\begin{bmatrix}
W^0 & W^0 & W^0 & \ldots & W^0 \\
W^0 & W^1 & W^2 & \ldots & W^N \\
W^0 & W^2 & W^4 & \ldots & W^{2N} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
W^0 & W(N-1) & W(2N-1) & \ldots & W(N-1)^2
\end{bmatrix}
\begin{bmatrix}
X_0(0) \\
X_0(1) \\
X_0(2) \\
X_0(N-1)
\end{bmatrix}
\]

A brief review of matrix properties can be found in Appendix A. Because of the periodic nature of \(W\) the following is true:

\[
W^{nk} \mod N = W^{nk}
\] (10)

\[
W^0 = 1 \text{ therefore, if } nk \mod N = 0, W^{nk} = 1
\]

This reduces the calculations as several of the \(W\) terms go to 1 and the highest power of \(W\) is \(N\). All of \(W\) values are complex, so most of the operations will have to be complex operations. We will continue to use only the \(W, X(n)\) and \(X(0(k))\) symbols to represent these complex quantities.

The FFT algorithm we will use requires that \(N\) be an integral power of 2. Other FFT algorithms do not have this restriction, but they are more complex to understand and develop. Additionally, for the relatively small values of \(N\) we are using this restriction should not provide much of a problem. We will define \(EXPONENT\) as \(\log_2 N\). Therefore,

\[
N = 2^{EXPONENT}
\]

The magic of the FFT, (as detailed in Appendix A), involves factoring the matrix into \(EXPONENT\) matrices, each of which has all zeros except for a 1 and a \(W^{nk}\) term in each row. When these matrices are multiplied together the result is the same as that of the multiplication indicated in Equation 9, except that the rows are interchanged and there are fewer non-trivial multiplications. To reorder the rows, and thus make the information useful, it is necessary to perform a procedure called "Bit Reversal".

This process requires that \(N\) first be converted to a binary number. The least significant bit (lsb) is swapped with the most significant bit (msb). Then the next lsb is swapped with the next msb, and so on until all bits have been swapped once. For \(N = 8\), 3 bits are used, and the values for \(N\) and their bit reversals are shown below:

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
<th>Bit Reversal</th>
<th>Decimal BR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

Recall that the FFT of real data provides a mirrored image output, but the FFT algorithm can accept inputs with both real and imaginary components. Since the inputs for harmonic analysis provided by a single A to D are real, the FFT algorithm is doing a lot of calculations with one input term equal to zero. This is obviously not very efficient. More information for a given size transform can be obtained by using a few more tricks.
It is possible to perform the FFT of two real functions at the same time by using the imaginary input values to the FFT for the second real function. There is then a post processing performed on the FFT results which separate the FFTs of the two functions. Using a similar procedure one can perform a transform on 2N real samples using an N complex sample transform.

The procedure involves alternating the real sample values between the real and imaginary inputs to the FFT. If, as in our example, the input to the FFT is a 2 by 32 array containing the complex values for 32 inputs, the 64 real samples would be loaded into it as follows:

| REAL       | 0 0 0 2 0 4 0 6 0 8 1 0 1 2 1 4 .... 6 0 6 2 |
| IMAGINARY  | 0 1 0 3 0 5 0 7 0 9 1 1 1 3 1 5 .... 6 1 6 3 |

This procedure is referred to as a pre-weave. In order to derive the desired results, the FFT is run, and then a post-weave operation is performed. The formula for the post-weave is shown below:

\[
X_r(n) = \left[ \frac{R(n)}{2} + \frac{R(N-n)}{2} \right] + \cos \frac{\pi n}{N} \left[ \frac{i(n)}{2} + \frac{i(N-n)}{2} \right] - \sin \frac{\pi n}{N} \left[ \frac{R(n)}{2} - \frac{R(N-n)}{2} \right] \quad n = 0, 1, \ldots, N-1
\]

\[
X_i(n) = \left[ \frac{i(n)}{2} - \frac{i(N-n)}{2} \right] - \sin \frac{\pi n}{N} \left[ \frac{R(n)}{2} - \frac{R(N-n)}{2} \right] \quad \cos \frac{\pi n}{N} \left[ \frac{i(n)}{2} + \frac{i(N-n)}{2} \right] \quad n = 0, 1, \ldots, N-1
\]

Where \( R(n) \) is the real FFT output value
\( I(n) \) is the imaginary FFT output value
\( X_r(n) \) is the real post-weave output
\( X_i(n) \) is the imaginary post-weave output

Note that the output is now one-sided instead of mirrored around the center frequency as it is in Figure 5. The magnitude of the signal at each frequency is calculated by taking the square root of the sum of the squares. The magnitude can now be plotted against frequency, where the frequency steps are defined as:

\[
\frac{n}{NT} \quad n = 0, 1, 2, 3, \ldots, N-1
\]

Where \( N \) is the number of complex samples (ie. 32 in this case) \( T \) is the time between samples.

A value of zero on the frequency scale corresponds to the DC component of the waveform. Most signal analysis is done using Decibels (dB), the conversion is \( \text{dB} = 10 \log_{10} (\text{Magnitude squared}) \). Decibels are not used as an absolute measure, instead signals are compared by the difference in decibels. If the ratio between two signals is 1:2 then there will be a 3 dB difference in their power.

5.0 USING THE FFT

There are several things to be aware of when using FFTs, but with the proper cautions, the FFT output can be used just like that of a spectrum analyzer. The
first precaution is that the FFT is a discrete approximation to a continuous Fourier Transform, so the output will seldom fit the theoretical values exactly, but it will be very close.

Since the programs in this application note generate a one-sided transform with \( N = 32 \), the frequency granularity is fairly course. Each of the frequency components output from the FFT is actually the sum of all energy within a narrow band centered on that frequency. This band of sensitivity is referred to as a “bin”. The reported magnitude is the actual magnitude multiplied by the value of the bin window at the actual frequency. Figure 6 shows several bin windows. Note that these windows overlap, so that a frequency midway between the two center frequencies will be reported as energy split between both windows. Be careful not to confuse the sampling window \( NT \) with bin windows or with the windowing function.

Another area of caution is the relationship of the sampling window to the frequency of the waveform. For the best accuracy, the window should cover an exact multiple of the period of the waveform being analyzed. If it covers less than one period, the results will be invalid. Other variations from ideal will not produce invalid results, just additional noise in the output.

If the sampling window does not cover an exact multiple of all of the frequency components of a waveform, the FFT results will be noisy. The reason for this is the sharp edge that the FFT sees when the edges of the window cut off the input waveform. Figure 7 shows a waveform that is an exact multiple of the window and

![Figure 7. Waveform is a Multiple of the Window](image)

![Figure 8. Waveform is Not a Multiple of the Window](image)
the periodic waveform that the FFT output reflects. In Figure 8, the waveform is not a multiple of the window and the waveform that the FFT output reflects has discontinuities. These discontinuities contribute to the noise in an FFT output. This noise is called "spectral leakage", or simply "leakage", since it is leakage between one frequency spectrum and another which is caused by digitization of an analog process.

To reduce this leakage, a process called windowing is used. In this procedure the input data is multiplied by specific values before being used in the FFT. The term "windowing" is used because these values act as a window through which the input data passes. If the input window goes smoothly to zero at both endpoints of the sampling window, there can be no discontinuities. Figure 9 shows a Hanning window and its effect on the input to an FFT. The Hanning window was named after its creator, Julius Von Hann, and is one of the most commonly used windows. More information on windowing and the types of windows can be found in the paper by Harris listed in the bibliography. As expected, the results of the FFT are changed because of the input windowing, but it is in a very predictable way.

Using the Hanning window results in bin windows which are wider and lower in magnitude than normal, as can be seen by comparing Figure 6 with Figure 10. For an input frequency which is equal to the center frequency of a bin window, the attenuation will be 6 dB on the center frequency. Since the bin windows are
wider than normal, the input frequency will also have
energy which falls into the bins on either side of center.
These side bins will show a reading of 6 dB below the
center window. The disadvantage of this spreading is
far less than the advantage of removing leakage from
the FFT output.

A set of FFT output plots are included in the Appen­
dix. These plots show the effect of windowing on vari­
ous signals. There are examples of all of the cases de­
scribed above. A brief discussion of the plots is also
presented.

Applications which can make use of this frequency
magnitude information include a wide range of signal
processing and detection tasks. Many of these tasks use
digital filtering and signature analysis to match signals
to a standard. This technique has been applied to anti­
knock sensors for automobile engines, object identifica­
tion for vision systems, cardiac arrhythmia detectors,
noise separation and many other applications. The abil­
ity to do this on a single-chip computer opens a door to
new products which would have not been possible or
cost effective previously.

The next four sections of this application note cover the
operation of the programs on a line by line basis. Section
6 shows an implementation of the FFT algorithm
in BASIC. This code is used as a template to write the
ASM96 code in Section 7. Sections 8, 9, and 10 cover
the code sections which support the FFT module. After
all of the code sections are discussed, an overview of
how to use the program is presented in Section 11.

6.0 BASIC PROGRAM FOR FFTS

The algorithm for this FFT is shown in the flowchart in
Figure 11 and the BASIC program in Listing 1. There
are four sections to this program: initialization, pre­
weaving, transform calculation, and post-weaving. The
flowchart is generalized, however, the BASIC program
has been optimized for assembly language conversion
with 64 real samples.

On the flowchart, the initialization and pre-weaving
sections are incorporated as “Read in Data”. The data
to be read includes the raw data as well as the size of
the array and the scaling factor. The details for pre­
weaving have been discussed earlier, and initialization
varies from computer to computer. LOOP COUNT
keeps track of which of the factored matrices are being
multiplied. SHIFT is the shift count which is used to
determine the power of W (as defined earlier) which
will be used in the loop.

For each loop N calculations are performed in sets of
two. Each calculation set is referred to as a butterfly
and has the following form:

```
Matrix L
X(k) + *Wp1 + X(k)
X(k+N2) + *Wp2 + X(k+N2)
```

Also Shown as:

```
X(k) X(k)
Wp1 Wp1
X(k+N2) X(k+N2)
```

OR

```
X1(k) = X0(k+N2)*Wp1 + X0(k)
X1(k+N2) = X0(k)*Wp2 + X0(k+N2)
```

In general, the W factors are not the same. However,
for the case of this FFT algorithm, Wp1 will always
equal (-Wp2). This is because of the way in which “p”
is calculated, and the fact that W(x) is a sinusoidal
function.

The inner loop in the flowchart is performed N2 times.
For LOOP = 1, N2 = N/2 and if INCNT = N2 then
k = N2 and k + N2 = N, so the first loop is done and
parameters LOOP, N2, and SHIFT are updated. For
the first loop, all N/2 sets of calculations are performed
contiguously. As LOOP increases, the number of con­
tiguous calculations are cut in half, until
LOOP = EXPONENT.

When LOOP = EXPONENT, N2 = 1, the butterfly is
then performed on adjacent variables. Figure 12 shows
the butterfly arrangement for a calculation where
N = 8, so that EXPONENT = 3.

The BASIC program follows this flowchart, but opera­
tions have been grouped to make it easier to convert it
to assembly language. Also not shown in the flowchart
are several divide by 2 operations. There are five in the
main section, one per loop. These provide the T/NT
factor in equation 8 for N = 32 (2^5 = 32). There is also
an extra divide by two in the post-weave section. It is
required to prevent overflows when performing the 16­
bit signed arithmetic in the ASM96 program. As a re­
sult of these operations, the input scale factor is ±1 =
±32767 and the output scaling is ±1 = ±16384.
Note, the maximum input values are ±0.99997.
Figure 11. Flowchart of Basic Program
Listing 1—BASIC FFT Program

```
100 ' THIS IS FFT13, FEBRUARY 4, 1986
105 ' COPYRIGHT INTEL CORPORATION, 1985
110 ' BY IRA HORDEN, MCO APPLICATIONS
115 ' 125 ' THIS PROGRAM PERFORMS A FAST FOURIER TRANSFORM ON 64 REAL DATA POINTS
130 ' USING A 2N-POINTS WITH AN N-POINT TRANSFORM ALGORITHM. THE FIRST
135 ' SECTION OF THE PROGRAM PERFORMS A STANDARD TRANSFORM ON DATA THAT HAS
140 ' BEEN INTERLEAVED BETWEEN THE REAL AND IMAGINARY INPUT VALUES. THE
145 ' RESULTS OF THAT TRANSFORM ARE THEN POST-PROCESSED IN THE SECOND SECTION
150 ' OF THE PROGRAM TO PROVIDE THE 32 OUTPUT BUCKETS. THE OUTPUT VALUES ARE
155 ' MULTIPLIED BY "M" TO MAKE IT EASY TO COMPARE WITH THE ASM-96 PROGRAM
160 ' 165 INPUT "NAME OF LIST FILE"; LST$
170 PRINT
175 OPEN LST$ FOR OUTPUT AS #1
180 ' 200 DIM XR(32),XI(32),WR(32),WI(32),SR(32)
210 N=32 : N1=31 : N2=N/2 ' N=NUMBER OF DATA POINTS
220 M=16383 ' M=MULT. FACTOR FOR SCALING
230 PI=3.141592654 : TPN=2*PI/N ' 2**E=N
240 FOR P=0 TO 31 : PN=P*TPN
250 WR(P)=COS(PN) : WI(P)=-SIN(PN) ' READ BR(P)
260 ' 270 FOR K=0 TO 31 ' READ IN DATA
280 FOR I=0 TO 31 ' READ XR(I)
290 INCNT=0 ' INITIALIZATION OF LOOP
300 IF LOOP>EXponent THEN 700
310 INCNT=INCNT+1 ' ACTUAL CALCULATIONS BEGIN HERE
320 XP=INT(K/(2^SHIFT)))
330 WRp=WR(P) : WIp=WI(P) : K2=K+N2 ' WRP AND WIP ARE CONSTANTS BASED ON
340 TMPR= (WRP*XR(K2) - WIP*XI(K2))/2 ' SINES AND COSINES OF BIT REVERSED
350 TMPI= (WRP*XI(K2) + WIP*XR(K2))/2 ' VALUES OF K SHIFITED RIGHT 5 TIMES
360 XR(K+K2) = TMPRI - TMPR ' TMPR, TMPI ARE THE REAL AND IMAGINARY
370 XI(K) = TMPII + TMPFR ' RESULTS OF A COMPLEX MULTIPLICATION
380 XR(K) = TMPRI + TMPFR
390 XI(K) = TMPII + TMPF1
400 ' 410 K=K+1
420 IF INCNT<N2 THEN GOTO 450 ' SINCE THE ARRAY IS PROCESSED 2 POINTS AT A TIME,
430 K=K+N2 ' ONLY N/2 LOOPS NEED TO BE MADE. ON EACH PASS,
440 LOOP=LOOP+1 : N2=N2/2 ' THE VALUE OF N2 CHANGES AND SMALLER CONSECUTIVE
450 IF INCNT=N2 THEN GOTO 430 ' SECTIONS ARE PROCESSED.
460 GOTo 400
```

270189-17
POST-PROCESSING AND REORDERING BEGIN HERE

FOR K = 0 TO 31
KPIN = K*PIN

XBRK = XI(BR(K))
XBRNK = XR(BR(N-K))

TI = (XBRK+XBRNK)/2
TR = (XBRK-XBRNK)/2

XRT = TI + TR*COS(KPIN)/2 - TR*SIN(KPIN)/2
XIT = TI - TR*COS(KPIN)/2 + TR*SIN(KPIN)/2

MAGSQ = OUTR*OUTR + OUTI*OUTI

MAG = SQR(MAGSQ)

IF MAGSQ*M < .5 THEN DECIBEL = 0 : GOTO 900

DBFACT = M/2/32767*M ' M^2 / 64K

DECIBEL = 10*LOG(MAGSQ*DBFACT)

DBCIBBL = .434294481

OUTR = XRT + TI*COS(KPIN)/2 - TR*SIN(KPIN)/2
OUTI = XIT - TI*SIN(KPIN)/2 + TR*COS(KPIN)/2

MAG = SQR(MAGSQ)

IF LST$ <> "SCRN:" THEN PRINT #1, CHR$(12)

END

DATA 0, 16, 8, 24, 4, 20, 12, 28, 2, 18, 10, 26, 6, 22, 14, 30
DATA 1, 17, 9, 25, 5, 21, 13, 29, 3, 10, 11, 27, 7, 23, 15, 31

DATA 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2

DATA 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2

DATA 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2

DATA 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2

Listing 1—BASIC FFT Program (Continued)
Lines 165-175 set up the file for printing the data, this can be SCRN:, LPT1:, or any other file.

The variables on line 470, TMP1 and TMPI1 would normally not be used in a BASIC program as more than one operation can be performed on each line. However, indirect table lookups always use a separate line of assembly code, so separate lines have been used here.

Lines 700-810 perform the post-weave. This is not in the flowchart, but can be found in Equation 11. Once again, table look-ups are separated and additional variables are used for clarity. The variables BR(x) are the bit reversal values of x.

Line 830 calculates the magnitude of the harmonic components.

Lines 900-950 print the results of the calculations, with line 900 determining if the print-out should be in hex or decimal.

Lines 1000-1080 are the data for the bit reversal values and input datapoints. The input waveform is one cycle of a square-wave.

**7.0 ASM96 PROGRAM FOR FFTS**

The BASIC program just presented has been used as an outline for the ASM96 program shown in Listing 2. There are many advantages to using the BASIC program as a model, the main ones being debugging and testing. Since the BASIC program is so similar in program flow to the ASM96 program, it's possible to stop the ASM96 program at almost any point and verify that the results are correct.
This module performs a fast fourier transform (FFT) on 64 real data points using a 2N-point algorithm. The algorithm involves using a standard FFT procedure for 32 real and 32 imaginary numbers. The real and imaginary arrays are filled alternately with real data points, and the output of the FFT is run through a post-processor. The result is a one sided array with 32 output buckets. The post processing includes a table lookup algorithm for taking the square root of an unsigned 32-bit number. 

All of the calculations in the main FFT program are done using 16-bit signed integers. The maximum value of any frequency component is therefore +/- 32K. (Note that a square wave of +/- 32K has a fundamental component greater than +/- 40K). Wherever possible tables are used to increase the speed of math operations. The complete transform, including obtaining the absolute magnitude of each frequency component, executes in 12 milliseconds with internal variables, 14 ms with external.

The program requires two 32-word input arrays, with the sample values alternated between the two. These start at XREAL and XIMAG. The resultant magnitude will be placed in a 32-word array at FFT_OUT. These are all externally defined variables. The external constant SCALE_FACTOR is used to divide the output when averaging will be used. Since the program averages its output, it is necessary to clear the array based at FFT_OUT before calling FFT_CALC to start the program.

The program was originally written in BASIC for testing purposes. The comments include these BASIC statements to make it easier to follow the algorithm.
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJEKT</th>
<th>LINK</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>39</td>
<td>RSCH</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>40</td>
<td>EXTN</td>
<td>portl, zero, error</td>
</tr>
<tr>
<td>0024</td>
<td>42</td>
<td>OSCH</td>
<td>at 24H</td>
</tr>
<tr>
<td>0024</td>
<td>43</td>
<td>TMF1</td>
<td>dal 1 : Temporary register, Real</td>
</tr>
<tr>
<td>0024</td>
<td>44</td>
<td>TMF2</td>
<td>dal 1 : Temporary register, Imaginary</td>
</tr>
<tr>
<td>0028</td>
<td>45</td>
<td>TMF1l</td>
<td>dal 1 : Temporary register, Real</td>
</tr>
<tr>
<td>0030</td>
<td>46</td>
<td>TMF2l</td>
<td>dal 1 : Temporary register, Imaginary</td>
</tr>
<tr>
<td>0034</td>
<td>47</td>
<td>XIMP</td>
<td>dal 1 : Temporary data register, Real</td>
</tr>
<tr>
<td>0038</td>
<td>48</td>
<td>XIRP</td>
<td>dal 1 : Temporary data register, Imaginary</td>
</tr>
<tr>
<td>0040</td>
<td>49</td>
<td>XIRK</td>
<td>dal 1</td>
</tr>
<tr>
<td>0044</td>
<td>50</td>
<td>XIRKX</td>
<td>dal 1</td>
</tr>
<tr>
<td>0046</td>
<td>51</td>
<td>XIRK</td>
<td>dal 1</td>
</tr>
<tr>
<td>0046</td>
<td>52</td>
<td>XIRKX</td>
<td>dal 1</td>
</tr>
<tr>
<td>0046</td>
<td>53</td>
<td>diff</td>
<td>equ xrrk :long ; Table difference for square root</td>
</tr>
<tr>
<td>0046</td>
<td>54</td>
<td>sqrt</td>
<td>equ xrrmk :long ; Square root</td>
</tr>
<tr>
<td>0046</td>
<td>55</td>
<td>log</td>
<td>equ xrrmk :long ; 10 Log magnitude^2</td>
</tr>
<tr>
<td>0046</td>
<td>56</td>
<td>nttloc</td>
<td>equ xirk :long ; Next location in table</td>
</tr>
<tr>
<td>0046</td>
<td>57</td>
<td>WRF</td>
<td>equ xrrk :word ; Multiplication factor, Real</td>
</tr>
<tr>
<td>0046</td>
<td>58</td>
<td>WFP</td>
<td>equ xrrk+2 :word ; Multiplication factor, Imaginary</td>
</tr>
<tr>
<td>0046</td>
<td>59</td>
<td>PMR</td>
<td>equ xrrk :word</td>
</tr>
<tr>
<td>0046</td>
<td>60</td>
<td>IN_CNT</td>
<td>equ xrrmk+2 :word</td>
</tr>
<tr>
<td>0046</td>
<td>61</td>
<td>NDBV2</td>
<td>equ xirk :word ; n divided by 2 (0 &lt; n &lt; N) #2</td>
</tr>
<tr>
<td>0046</td>
<td>62</td>
<td>KPTP</td>
<td>daw 1 ; K for counter #2 to index words</td>
</tr>
<tr>
<td>0046</td>
<td>63</td>
<td>KEX</td>
<td>daw 1 ; KPTP + NDIV2</td>
</tr>
<tr>
<td>0046</td>
<td>64</td>
<td>N_SUB_K</td>
<td>daw 1 ; N- #2 to index words</td>
</tr>
<tr>
<td>0046</td>
<td>65</td>
<td>HX</td>
<td>daw 1 ; Bit reversed pointer of KPTP</td>
</tr>
<tr>
<td>0046</td>
<td>66</td>
<td>HXK</td>
<td>daw 1 ; Bit reversed pointer of N_SUB_K</td>
</tr>
<tr>
<td>0046</td>
<td>67</td>
<td>SHFP_CNT</td>
<td>daw 1</td>
</tr>
<tr>
<td>0046</td>
<td>68</td>
<td>LOOP_CNT</td>
<td>daw 1</td>
</tr>
<tr>
<td>0046</td>
<td>69</td>
<td>ptr</td>
<td>equ km2 :word ; Pointer for square root table</td>
</tr>
<tr>
<td>0000</td>
<td>70</td>
<td>DSHG</td>
<td></td>
</tr>
<tr>
<td>0048</td>
<td>71</td>
<td>EXTN</td>
<td>FFT_MODE : mode for FFT input and graphing</td>
</tr>
<tr>
<td>0048</td>
<td>72</td>
<td>EXTN</td>
<td>XREAL, XIMAG : Base addresses for 32 16-bit signed</td>
</tr>
<tr>
<td>0048</td>
<td>73</td>
<td>EXTN</td>
<td>FFT_OUT : Starting address for 32 word array</td>
</tr>
<tr>
<td>0048</td>
<td>74</td>
<td>EXTN</td>
<td>entries for real and imaginary numbers respectively.</td>
</tr>
<tr>
<td>0048</td>
<td>75</td>
<td>EXTN</td>
<td>FFT_OUT : Starting address for 32 word array</td>
</tr>
<tr>
<td>0048</td>
<td>76</td>
<td>EXTN</td>
<td>of magnitude information.</td>
</tr>
<tr>
<td>0048</td>
<td>77</td>
<td>EXTN</td>
<td>FFT_OUT : Starting address for 32 word array</td>
</tr>
<tr>
<td>0048</td>
<td>78</td>
<td>EXTN</td>
<td>of magnitude information.</td>
</tr>
<tr>
<td>0000</td>
<td>79</td>
<td>EXTN</td>
<td>FFT_OUT : Starting address for 32 word array</td>
</tr>
<tr>
<td>0040</td>
<td>80</td>
<td>OUTI</td>
<td>daw 32 ; Real component of fft</td>
</tr>
<tr>
<td>0040</td>
<td>81</td>
<td>OUTI</td>
<td>daw 32 ; Imaginary component of waveform</td>
</tr>
<tr>
<td>0040</td>
<td>82</td>
<td>PUBLIC</td>
<td>OUTI, OUTI</td>
</tr>
<tr>
<td>0040</td>
<td>83</td>
<td>PUBLIC</td>
<td>OUTI, OUTI</td>
</tr>
<tr>
<td>0040</td>
<td>84</td>
<td>$REJCT</td>
<td></td>
</tr>
</tbody>
</table>
MCS-96 MACRO ASSEMBLER  FFT_RUN

ERR LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

ERR  LOC  OBJECT  LINE  SOURCE STATEMENT

02/18/86  PAGE  3

Listing 2—ASM96 FFT Program (Continued)

6-120

270189-35
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>228E</td>
<td>F44F0003C4</td>
<td>130</td>
<td>; ; Complex multiplication follows</td>
</tr>
<tr>
<td>22CD</td>
<td>F44F0003B2</td>
<td>132</td>
<td>mul  tmp1, wrp, xreal[kn2]</td>
</tr>
<tr>
<td>22CF</td>
<td>F44F0003C4</td>
<td>134</td>
<td>sub  tmp1+2, tmp1+2</td>
</tr>
<tr>
<td>22DF</td>
<td>F44F0003B2</td>
<td>136</td>
<td>mul  tmp1, wpi, xreal[kn2]</td>
</tr>
<tr>
<td>22DC</td>
<td>5442EA</td>
<td>138</td>
<td>add  tmp1+2, tmp1+2</td>
</tr>
<tr>
<td>22D0</td>
<td>DC55</td>
<td>140</td>
<td>; ; using the high byte only of a signed multiply</td>
</tr>
<tr>
<td>22EC</td>
<td>A34D0002C</td>
<td>144</td>
<td>ld    tmp1, xreal[kptr]</td>
</tr>
<tr>
<td>22FF</td>
<td>A0112C</td>
<td>146</td>
<td>shra  tmp1, #1</td>
</tr>
<tr>
<td>22EF</td>
<td>A01130</td>
<td>148</td>
<td>shra  tmp1, #1</td>
</tr>
<tr>
<td>22F2</td>
<td>442523C4</td>
<td>150</td>
<td>gr2: sub xtmp, tmp1, tmp1+2</td>
</tr>
<tr>
<td>22FB</td>
<td>C34000034</td>
<td>152</td>
<td>xtmp, xreal[kn2]</td>
</tr>
<tr>
<td>22FC</td>
<td>442A3038</td>
<td>154</td>
<td>xtmp, ximag[kn2]</td>
</tr>
<tr>
<td>2304</td>
<td>442623C4</td>
<td>156</td>
<td>add  xtmp, tmp1, tmp1+2</td>
</tr>
<tr>
<td>2308</td>
<td>444D00034</td>
<td>158</td>
<td>xtmp, xreal[kptr]</td>
</tr>
<tr>
<td>230D</td>
<td>44243038</td>
<td>160</td>
<td>xtmp, ximag[kptr]</td>
</tr>
<tr>
<td>2311</td>
<td>C34000034</td>
<td>162</td>
<td>; ; using the high byte only of a signed multiply</td>
</tr>
<tr>
<td>2316</td>
<td>DC23</td>
<td>164</td>
<td>$execute</td>
</tr>
</tbody>
</table>

Comments:

- Line 130: Branch on error in complex multiplications
- Line 151: Provides an effective divide by two
- Line 156: Branch on error in complex additions
MCS-96 MACRO ASSEMBLER  FFT_RUN

ERR LOC OBJECT   LINE   SOURCE STATEMENT

2316 6502004C   165 ;   560 K=K+1
       166 ik:    add  kptr,#2
       167
       168 ;   570 IF INCNT=K2 THEN GOTO 450
231C 004442   169   cmp  in_cnt,ndiv2
       170      bit  IN_LOOP
       171
       172 ;   580 K=K+N2
231F D6022778   173   add  kptr,ndiv2
       174 ;   590 IF K=N1 THEN GOTO 430
2323 64444C   175   cmp  kptr,#62
       176      bit  MID_LOOP
       177
       178 ;   600 LOOP=LOOP+1 ; N2=N2/2
232E 1758   179   incb  loop_cnt
       180      shift loop_cnt  ; 605 SHIFT=SHIFT+1
2333 1556   181   decb  shift_cnt
       182 ;   610 GOTO 400
       183
       184
       185
       186   E 186  ERR1:    ldb  error,#01  ; overflow error, 1st set of calculations
       187       ret
233A F0   188   ERR2:    ldb  error,#02  ; overflow error, 2nd set of calculations
       189       ret
       190
       191  $EXIT

02/18/86  PAGE 5

Listing 2—ASM96 FFT Program (Continued)

270189-37
233F 014C 197 ; 720 FOR K=0 TO 31
233F 014D 198
2344 A35000003C 201 int xrrk,xxreal[krk]
2344 A35000044 202 id xirx,xximag[krk]
2348 A35000044 203
2352 A35000050 204
2355 A35000044 205
2372 A44226 215 ld tmpl+2,xirk+2
2377 080128 217 ; 16 bit result in tmpl
2388 44403C34 220
2395 44404C34 225 ; 780 XOR= (XBRK + XBRK)/4
2397 A44226 222 subc tmp+2,xrrk+2
2398 080128 223 ; 16 bit result in tmpr
2399 A03E26 224
2399 A03E26 225
2399 A44226 226 addc xrrk,xxreal[krk]
2399 A44226 227
2399 A44226 228
2399 000E34 229 ; 32 bit result in xrtmp
2399 A03E26 230
2399 A03E26 231
2399 A44226 232
2399 A44226 233
2399 A44226 234
2399 000E34 235 ; 32 bit result in xitmp
2399 000E34 236
239F 000E38 237
Multiply will provide effective divide by 2

```
2380 OUTR = (XIT + T*SINF(K)/2 - T*COSF(K)/2)
```

```
2381 OUTI = (XIT - T*SINF(K)/2 - T*COSF(K)/2)
```

```
2382 MAG = SQRT(OUTR^2 + OUTI^2)
```

```
2383 GET_MAG: ; Get Magnitude of Vector
```

```
2384 FFT_MODE,2,CALC_SQRT
```

```
2385 $eject
```
; ; ; *** CALCULATE 10 log magnitude;2 ***
; Output = 512*10*LOG(x) x=1,2,3 ... 64K
276
277
278
279
23F7
23F7 0156
23F9 0F5624
23FC 990F56
23FF DA04
2401 0140
2403 202C
2405 44565656
2409 AC274E
2410 660E314E
2414 A24F40
2417 A24E44
241A 68A404
241D AC253C
2420 6C443C
2423 0C8R3C
2426 64C440
2429 080540
242C A7570A3C40
2431 0B0040
2434 A40040
2437 6740000040
243C C340000040
2441 2045
244B 0189-40
; ; *** CALCULATE SQUARE ROOT ***
2443 0156  clr  shft_cnt
2445 0F5624  normal  tmp,r,shft_cnt ; Normalize and get normalization factor
2448 9705  jne  SQRT_IN_RANGE ; Jump if tmp > 0
244A C04200  zero, sqrt+2
244B 2029  br  SQRT_STORE

244F 390  SQRT_IN_RANGE:
244F AC74E  ldFIXe  ptr,tmp+3 ; Most significant byte is table pointer
2452 444E4E4E  add  ptr,ptr,ptr
2456 6606394E  add  ptr, # SQ_TABLE-256 ; ptr= Table + offset (offset=tmp+3)
245A A24F40  ld  sqrt,[ptr]+
245B A24B44  ld  nxloc,[ptr] ; Linear Interpolation
2460 684044  sub  nxloc,sqrt ; nxloc = sqrt - next sqrt
2463 AC263C  ldFIXe  diff,tmp+2 ; diff=1 = nxloc * sqrt+2 / 256
2466 6C443C  mulu  diff,nxloc
2469 AC383C  ldFIXe  diff,diff+1 ; sqrt = sqrt + delta (diff < OFFFH)
246C 643C40  add  sqrt,diff
246F 44665656  add  shft_cnt,shft_cnt,shft_cnt
2473 6F57C83940  mulu  sqrt,tab_sqrt[shft_cnt] ; divide by normalization factor
2476 390  ; ; ; mulu acts as divide since if tab2=OFFFH
2479 350  ; ; ; sqrt would remain essentially unchanged

247B 0B0042  shr  sqrt+2,#SCALE_FACTOR
247B A0042  addc  sqrt+2,zero ; Divide to prevent overflow during
247F 674B00042  add  sqrt+2,FFT_OUT[kptr] ; averaging of outputs
2483 C34B00042  at  sqrt+2,FFT_OUT[kptr]
2486 358  ; ; ; *** END OF LOOP ***
248B 6502004C  add  kprr,#2
248C 69020050  sub  n_sub_k,#2
2490 DF0226B4  bne  UN_LOOP
2493 FF  bnt
2494 F0  ; ; ;
2496 #execute
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>ERR</th>
<th>LOC OBJECT</th>
<th>SOURCE STATEMENT</th>
<th>LINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>367</td>
<td>&quot;nolist&quot;</td>
<td>368</td>
<td>CSEG AT 3600H</td>
<td>; ; ; ; Use 2k for tables</td>
<td>369</td>
</tr>
<tr>
<td>370</td>
<td>BRV:</td>
<td>2bit reversal value</td>
<td>371</td>
<td></td>
<td></td>
</tr>
<tr>
<td>372</td>
<td>DCW</td>
<td>240, 216, 248, 2424, 244, 2x20, 2x12, 2x28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>373</td>
<td>DCW</td>
<td>24, 218, 240, 2x26, 2x22, 2x14, 2x30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>374</td>
<td>DCW</td>
<td>2x1, 2x17, 2x24, 2x25, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>375</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>376</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>377</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>378</td>
<td>DCW</td>
<td>2x26, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>379</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>380</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>381</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>382</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>383</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>385</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>387</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>388</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>389</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>390</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>391</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>392</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>393</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>394</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>395</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>396</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>397</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>398</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>399</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>401</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>402</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>403</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>404</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>405</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>406</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>407</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>408</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>409</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410</td>
<td>DCW</td>
<td>2x10, 2x5, 2x21, 2x13, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>411</td>
<td>DCW</td>
<td>2x3, 2x19, 2x11, 2x27, 2x23, 2x15, 2x21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>412</td>
<td>DCW</td>
<td>2x2, 2x28, 2x25, 2x20, 2x22, 2x14, 2x29</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MCS-96 MACRO ASSEMBLER

ERR LOC OBJECT LINE SOURCE STATEMENT

3018 FFFF04B500000825A
3019 DCW 65535, 46340, 32768, 0, 21310, 16384, 11585, 8192, 5793
301A DCW 4096, 2986, 2048, 1449, 1024, 724, 512, 362
301B DCW 4096, 2896, 2048, 1448, 1024, 724, 512, 362
301C DCW 256, 181, 128, 91, 64, 45, 32, 23
301D DCW 16, 11, 8, 6, 4, 3, 2, 1

3020 SQ_TABLB: ; square root of n
3021 DCW 46341, 46522, 46702, 46881, 47059, 47237, 47415, 47591
3022 DCW 47767, 47942, 48117, 48291, 48465, 48637, 48809, 48981
3023 DCW 49152, 49322, 49492, 49661, 49830, 49998, 50166, 50332
3024 DCW 50499, 50665, 50830, 50995, 51159, 51323, 51486, 51649
3025 DCW 51811, 51972, 52134, 52294, 52454, 52614, 52773, 52932
3026 DCW 53090, 53246, 53405, 53562, 53719, 53874, 54030, 54185
3027 DCW 54340, 54494, 54647, 54801, 54954, 55106, 55258, 55410
3028 DCW 55561, 55712, 55862, 56012, 56162, 56311, 56469, 56618
3029 DCW 56765, 56915, 57061, 57208, 57344, 57490, 57636, 57781
302A DCW 57926, 58071, 58215, 58359, 58503, 58646, 58789, 58931
302B DCW 59073, 59215, 59357, 59498, 59639, 59779, 59919, 60059
302C DCW 60199, 60339, 60477, 60615, 60753, 60890, 61029, 61166
302D DCW 61309, 61440, 61576, 61712, 61848, 62198, 62220, 62364
302E DCW 62388, 62523, 62657, 62790, 62924, 63057, 63190, 63323
302F DCW 63465, 63607, 63749, 63890, 63592, 64113, 64243, 64374
3030 DCW 64508, 64649, 64792, 64933, 65074, 65215, 65356, 65498

270189-43
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3800</td>
<td>LOG_TABLE:</td>
<td>446</td>
<td>16384<em>10</em>LOG(2**(4-n)) n=128,129,130 ... 256</td>
</tr>
<tr>
<td>3818</td>
<td>DCW 0, 554, 1103, 1648, 2190, 2727, 3260, 3789</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3828</td>
<td>DCW 3514, 3436, 5353, 5866, 6376, 6883, 7386, 7889</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3838</td>
<td>DCW 8381, 8873, 9362, 9848, 10330, 10810, 11286, 11758</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3848</td>
<td>DCW 12228, 12695, 13158, 13619, 14076, 14531, 14983, 15432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3858</td>
<td>DCW 15876, 16321, 16762, 17200, 17635, 18067, 18497, 18925</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3868</td>
<td>DCW 19349, 19772, 20191, 20609, 21024, 21436, 21846, 22254</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3878</td>
<td>DCW 22660, 23063, 23464, 23862, 24259, 24653, 25045, 25435</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3888</td>
<td>DCW 25822, 26208, 26592, 26973, 27353, 27730, 28106, 28479</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3898</td>
<td>DCW 28851, 29220, 29588, 29954, 30316, 30680, 31040, 31399</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3908</td>
<td>DCW 31755, 32110, 32463, 32815, 33165, 33512, 33859, 34203</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3918</td>
<td>DCW 34546, 34897, 35227, 35556, 35882, 36236, 36570, 36901</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3928</td>
<td>DCW 37232, 37560, 37887, 38213, 38537, 38860, 39181, 39501</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3938</td>
<td>DCW 39819, 40136, 40452, 40766, 41079, 41390, 41700, 42009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3948</td>
<td>DCW 42316, 42622, 42927, 43230, 43533, 43833, 44133, 44431</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3958</td>
<td>DCW 44729, 45024, 45319, 45612, 45905, 46196, 46486, 46774</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3968</td>
<td>DCW 47062, 47348, 47633, 47917, 48200, 48482, 48763, 49042</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3978</td>
<td>DCW 49321</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LOG_OFFSET: 512*10*LOG(2**(15-n)) n=0,1,2,3 ... 15

LOG_OFFSET: 512*10*LOG(0.5) n=16,17,18 ... 31

3COA 4FBA4A544543F48 472 0 23119, 21578, 20037, 18495, 16854, 15413, 13871, 12330
3C1A 252A20241A161518 473 0 10769, 9249, 7726, 6165, 4624, 3083, 1541, 0
3C2A 475 END

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.
The BASIC program is used as comments in the ASM96 program. Some of the variables in the ASM96 program have slightly different names than their counterparts in the BASIC program. This was to make the comments fit into the ASM96 code. Highlights in this section of code are a table driven square root routine and log conversion routine which can easily be adapted for use by any program.

Both the square root routine and the log conversion routine use the 32-bit value in the variable TMR. The square root routine calculates the square root of that value in the variable SQRT+2, a 16-bit variable. In this program, the square root value is averaged and stored in a table.

The log conversion routine divides the value in TMR by 65536 \((2^{16})\) and uses table lookup to provide the common log. The result is a 16-bit number with the value \(512 \times 10 \log (\text{TMR}/65536)\) stored in the variable LOG. This calculation is used to present the results of the FFT in decibels instead of magnitude. With an input of 63095, the output is \(512 \times 48\) dB. The graph program, (Section 10), prints the output value of the plot as INPUT/512 dB.

The following descriptions of the ASM code point out some of the highlights and not-so-obvious coding:

Lines 1-104 initialize the code and declare variables. The input and output arrays of the program are declared external. Note that many of the registers are overlayable, use caution when implementing this routine with others with overlayable registers.

Lines 116–124 calculate the power of W to be used. Note that KPTR is always incremented by 2. The multiple right shift followed by the AND mask creates an even address and the indirect look to the BR (Bit Reversal) table quickly calculates the power PWR.

Lines 130–138 perform the complex multiplications. Since WIP and WRP range from \(-32767\) to \(+32767\), the multiplication is easy to handle. The automatic divide by two which occurs when using the upper word only of the 32-bit result is a feature in this case.

Lines 144–163 use right shifts for a fast divide, then add or subtract the desired variables and store them in the array. Note that the upper word of TMR and TMI is used, and the same array is used for both the input and output of the operations.

Lines 165–189 update the loop variables and then check for errors on the complex multiplications and additions. If there are no overflows at this time the data will run smoothly through the rest of the program.

Lines 200–212 load variables with values based on the bit reversed values of pointers.

Lines 214–236 perform additions and subtractions to prepare for the next set of formulas. Note that XITMP and XRTMP are 32-bit values.
Lines 240-260 perform multiplies and summations resulting in 32-bit variables. This saves a bit or two of accuracy. The upper words are then stored as the results.

Lines 263-272 generate the squared magnitude of the harmonic component as a 32-bit value.

Lines 278-310 calculate $10 \log(TMPR/65536)$. The 32-bit register TMPR is divided by 65536 so that the output range would be reasonable.

First, the number is normalized. (It is shifted left until a 1 is in the most significant bit, the number of shifts required is placed in SHFT_CNT.) If it had to be shifted more than 15 times the output is set to zero.

Next, the most significant BYTE is used as a reference for the look-up table, providing a 16-bit result. The next most significant BYTE is then used to perform linear interpolation between the referenced table value and the one above it. The interpolated value is added to the directly referenced one.

The 16-bit result of this table look-up and interpolation is then added to the Log of the normalization factor, which is also stored in a table. This table look-up approach works fast and only uses 320 bytes of table space. The results are valid to near 14-bits, more than enough for the FFT algorithm.

Lines 352-360 average the magnitude value, if multiple passes are being performed, and then store the value in the array. The loop-counters are incremented and the process repeats itself.

This concludes the FFT routine. In order to use it, it must be called from a main program. The details for calling this routine are covered in the next section.

8.0 BACKGROUND CONTROL PROGRAM

The main routine is shown in Listing 3. It begins with declarations that can be used in almost any program. Note that these are similar, but not identical, to other 8096 include files that have been published. Comments on controlling the Analog to Digital converter routine follow the declarations.
This program performs an FFT on real data and plots it on a printer.

It uses the program modules AD2CON, PLOTSP, and FFTRUN. The adjustable parameters of each of the programs are set by this main module.

$INCLUDE ("FO:DEMO6.INC") ; Include SFR definitions
$no118t ; Turn listing off for include file
******************************************************************************
Copyright 1985, Intel Corporation  
by Ira Horden, MOO Applications

$INCLUDE ("FO:DI!M096.INC") ; DEFINITION OF SYMBOLIC NAMES FOR THE I/O REGISTERS OF THE 8096
******************************************************************************
 ZERO AD_ClMWID
 AD RESULT  LO
 AD=RflSULT=HI
 RBI !IODB
 RBO-TIMB
 RBI=STATUS
 SBUF INT_lIASK
 INT_PBNOING
 SPSTAT
 IFATCIIDOG
******************************************************************************
270189-45
MCS-96 MACRO ASSEMBLER

PUBLIC ZERO, AD_COMMAND, AD_RESULT_LO, AD_RESULT_HI, HSI_MODE, HSI_TIME, HSI_TIME

PUBLIC HSI_COMMAND

PUBLIC HSI_STATUS, SBUP, INT_MASK, INT_PENDING, WATCHDOG, TIMER1, TIMER2

PUBLIC BAUD_REG, PORT0, PORT1, PORT2, SPSTAT, SMCR, I/O, I/O1, I/O2, I/O3

PUBLIC PWM_CONTROL, SP, CR, LF

PUBLIC

; Temp registers used in conformance with PD1-96 conventions.

; Busy

; Number of times to average the waveform

; a2d utility commands/responses for "control_a2d"

00007
0010
0028

; Convert to BurFO as PAINED SIGNED data

; Download Buff0
SCALE_FACTOR equ 0 ; Number of rights shifts performed on output of FFT. Used to prevent overflow on summation.

PLOT_HRES equ 256 ; Number of input units per plot unit

PLOT_HRES_2 equ plot_res/2

PLOT_MAX equ plot_res+145 ; 145 chars/row

PUBLIC scale_factor, plot_res, plot_res_2, plot_max

OSSEG at 24H ; common osseg area

tmpreal: ds1 1
tmpimag: ds1 1

vardptr: dsw 1

RSSEG

fft_mode: daw 1

error: daw 1

even: ds2 1

PUBLIC error, fft_mode

EXTERNAL sample_period, control_a2d

DSSEG at 80H

XREAL:

BEST_BUFF_BASE: DSW 64 ; For FFT routine

IMAG equ XREAL+64 ; For FFT routine

PUBLIC BEST_BUFF_BASE, XREAL, XIMAG

DSSEG at 200H

0200

PUBLIC FOR BUFFO_BASE, FFT_OUT, PLOT_IN

$}ject
MCS-96 MACRO ASSEMBLER  FFT_MAIN_APNOTE  02/18/86  PAGE 4

ERR LOC  OBJECT   LINE   SOURCE STATEMENT

2080  A1000018  R  141  EXTRN INIT_OUTPUT, DRAW_GRAPH, CON_OUT  ; For Plot Routine
2084  A30100301C  R  142  EXTRN FFT_CALC  ; For FFT routine
2089  010FD  R  143  EXTRN A2D_BUFF_UTIL  ; For A2D routine

2090  B100002  R  150  SBB a1, a1_wait  ; WAIT FOR SBE TO CLEAR SERIAL PORT INTERRUPTS
2094  B100020  R  151  SBB A1, a1_wait
2098  0120  R  152  SBB B1, a1_wait
209A  0125  R  153  SBB C1, a1_wait
209C  000020  R  154  SBB D1, a1_wait
209F  000020  R  155  SBB E1, a1_wait

20A1  00004  R  156  SBB avr_cnt, avr_num
20A4  2919  R  157  SBB LOAD_DATA
20A6  2902  R  158  SBB LOAD_DATA

20A8  282F  R  159  SBB LOAD_DATA
20BA  282F  R  160  SBB LOAD_DATA

20BC  282F  R  161  SBB LOAD_DATA
20BE  282F  R  162  SBB LOAD_DATA
20BF  282F  R  163  SBB LOAD_DATA

20C0  282F  R  164  SBB LOAD_DATA
20C2  282F  R  165  SBB LOAD_DATA
20C4  282F  R  166  SBB LOAD_DATA

20C6  282F  R  167  SBB LOAD_DATA
20C8  282F  R  168  SBB LOAD_DATA
20CA  282F  R  169  SBB LOAD_DATA

20CC  282F  R  170  SBB LOAD_DATA
20CE  282F  R  171  SBB LOAD_DATA
20D0  282F  R  172  SBB LOAD_DATA

20D2  282F  R  173  SBB LOAD_DATA
20D4  282F  R  174  SBB LOAD_DATA
20D6  282F  R  175  SBB LOAD_DATA
20D8  282F  R  176  SBB LOAD_DATA

20DA  282F  R  177  SBB LOAD_DATA
20DC  282F  R  178  SBB LOAD_DATA

20EE  282F  R  179  SBB LOAD_DATA

For Plot Routine
For FFT routine
For A2D routine

BEGIN: CALL INIT_OUTPUT  ; Initialize serial port
BEGIN: CALL INIT_OUTPUT  ; Initialize serial port
BEGIN: CALL INIT_OUTPUT  ; Initialize serial port

NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
NEW_TRANSFORM_SET:
### SOURCE STATEMENT

<table>
<thead>
<tr>
<th>Err</th>
<th>Loc</th>
<th>Obj</th>
<th>Line</th>
<th>Source Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>20C7</td>
<td>181</td>
<td>LOAD_DATA:</td>
<td>;**** LOAD DATA INTO RAM</td>
<td></td>
</tr>
<tr>
<td>20C7</td>
<td>182</td>
<td>1db</td>
<td>port1,#00</td>
<td>;**** FOR INDICATION ONLY</td>
</tr>
<tr>
<td>20C7</td>
<td>184</td>
<td>SET_A2D:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20C8</td>
<td>186</td>
<td>1db</td>
<td>control_a2d,#con_b0</td>
<td>Set converter for buffer0</td>
</tr>
<tr>
<td>20C9</td>
<td>187</td>
<td>orb</td>
<td>control_a2d,#01</td>
<td>Convert channel 1</td>
</tr>
<tr>
<td>20DC</td>
<td>188</td>
<td>1d</td>
<td>sample_period,#50</td>
<td>100 us sample period</td>
</tr>
<tr>
<td>20DE</td>
<td>190</td>
<td>CALL</td>
<td>#d2d_buff'util</td>
<td>Start the conversion process</td>
</tr>
<tr>
<td>20E0</td>
<td>190</td>
<td>1d</td>
<td>control_a2d,busy,</td>
<td></td>
</tr>
<tr>
<td>20DA</td>
<td>192</td>
<td>1db</td>
<td>control_a2d,#damp_b0_p_s</td>
<td>download b0 paired/signed</td>
</tr>
<tr>
<td>20DD</td>
<td>194</td>
<td>CALL</td>
<td>#d2d_buff'util</td>
<td></td>
</tr>
<tr>
<td>20DD</td>
<td>195</td>
<td>1db</td>
<td>control_a2d,#damp_b0_p_s</td>
<td>download b0 paired/signed</td>
</tr>
<tr>
<td>20DF</td>
<td>196</td>
<td>1db</td>
<td>control_a2d,#damp_b0_p_s</td>
<td>download b0 paired/signed</td>
</tr>
</tbody>
</table>

### TABLE LOAD:

<table>
<thead>
<tr>
<th>DATA0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>32767</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>-32767</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>-32767</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>-32767</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>-32767</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA5:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>-32767</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>-32767</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA7:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCW</td>
</tr>
<tr>
<td>-32767</td>
</tr>
</tbody>
</table>
### MCS-96 MACRO ASSEMBLER

<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2182</td>
<td></td>
<td>223</td>
<td>DO_WINDOW:</td>
</tr>
<tr>
<td>2182</td>
<td>O12C</td>
<td>224</td>
<td>clr wndptr</td>
</tr>
<tr>
<td>2184</td>
<td>O12E</td>
<td>225</td>
<td>clr varptr</td>
</tr>
<tr>
<td>2186</td>
<td></td>
<td>226</td>
<td>WINDOW:</td>
</tr>
<tr>
<td>2188</td>
<td>A5208E211C</td>
<td>227</td>
<td>ld ex, hanning[wndptr]</td>
</tr>
<tr>
<td>2189</td>
<td>A520C2120</td>
<td>228</td>
<td>ld bx, hanning2[wndptr]</td>
</tr>
<tr>
<td>2190</td>
<td>FE42F9B001C24</td>
<td>229</td>
<td>mul tmpreal, ex, xreal[varptr]</td>
</tr>
<tr>
<td>2197</td>
<td>FE42FC0002028</td>
<td>230</td>
<td>mul tmpimag, bx, ximag[varptr]</td>
</tr>
<tr>
<td>219E</td>
<td>000124</td>
<td>231</td>
<td>shll tmpreal, #1</td>
</tr>
<tr>
<td>21A1</td>
<td>000128</td>
<td>232</td>
<td>shll tmpimag, #1</td>
</tr>
<tr>
<td>21A4</td>
<td>C32FB00026</td>
<td>233</td>
<td>at tmpreal+2, xreal[varptr]</td>
</tr>
<tr>
<td>21A9</td>
<td>C32FC0002A</td>
<td>234</td>
<td>at tmpimag+2, ximag[varptr]</td>
</tr>
<tr>
<td>21AE</td>
<td>6504002C</td>
<td>235</td>
<td>add wndptr, #4</td>
</tr>
<tr>
<td>21B2</td>
<td>6502002E</td>
<td>236</td>
<td>add varptr, #2</td>
</tr>
<tr>
<td>21B8</td>
<td>8940002E</td>
<td>237</td>
<td>cmp varptr, #64</td>
</tr>
<tr>
<td>218A</td>
<td>D7CA</td>
<td>238</td>
<td>jne window</td>
</tr>
<tr>
<td>21BE</td>
<td>F0</td>
<td>239</td>
<td>RET</td>
</tr>
<tr>
<td>21B8</td>
<td></td>
<td>240</td>
<td>HANNING:</td>
</tr>
<tr>
<td>21BE</td>
<td>0004F003B01C102</td>
<td>241</td>
<td>DCM 0, 79, 315, 705, 1247, 1935, 2761, 3719</td>
</tr>
<tr>
<td>21CE</td>
<td>B126617711E0421</td>
<td>242</td>
<td>DCM 4799, 5990, 7281, 8660, 10114, 11628, 13187, 14778</td>
</tr>
<tr>
<td>21DE</td>
<td>00405467C0C45952</td>
<td>243</td>
<td>DCM 1538, 17989, 19580, 21139, 22653, 24107, 25646, 27677</td>
</tr>
<tr>
<td>21EE</td>
<td>40607F136757078</td>
<td>244</td>
<td>DCM 27968, 29048, 30006, 30832, 31520, 32062, 32452, 32888</td>
</tr>
<tr>
<td>21FE</td>
<td>F7FFFD7FC479E387D</td>
<td>245</td>
<td>DCM 32767, 32688, 32452, 32062, 31520, 30832, 30006, 29048</td>
</tr>
<tr>
<td>220E</td>
<td>406F0908B962B3E8</td>
<td>246</td>
<td>DCM 27968, 26777, 25486, 24107, 22653, 21139, 19580, 17989</td>
</tr>
<tr>
<td>221E</td>
<td>00405467C0C4223E2D</td>
<td>247</td>
<td>DCM 1538, 14778, 13187, 11628, 10114, 8660, 7281, 5990</td>
</tr>
<tr>
<td>222E</td>
<td>B12670EC900A907</td>
<td>248</td>
<td>DCM 4799, 3715, 2761, 1935, 1247, 705, 315, 79</td>
</tr>
<tr>
<td>223E</td>
<td>00000</td>
<td>249</td>
<td>DCM 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>223E</td>
<td></td>
<td>250</td>
<td>HANNING:</td>
</tr>
<tr>
<td>223E</td>
<td></td>
<td>251</td>
<td>DCM 4799, 3715, 2761, 1935, 1247, 705, 315, 79</td>
</tr>
</tbody>
</table>

### Note

- **PERFORM HANNING WINDOW**
- Windowing provides an effective divide by 2 because of the multiply
- Compensate for the divide by 2
- Windowing function

---

**Listing 3—Main Routine (Continued)**

Page 270189-50
### MCM-96 Macro Assembler

#### FFT_MAIN_APNOTB

<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>CSEB AT 3000H</td>
<td>; ADDITIONAL TABLES FOR TESTING</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>0000351E07928770</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>3010</td>
<td>A8A574F31C4777A7</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>3020</td>
<td>01980F5BBB8563C</td>
<td>261</td>
<td></td>
</tr>
<tr>
<td>3030</td>
<td>01A59F5B026508A</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>3040</td>
<td>0000CABB7706128F</td>
<td>263</td>
<td></td>
</tr>
<tr>
<td>3050</td>
<td>0255ABC0E480B085</td>
<td>264</td>
<td></td>
</tr>
<tr>
<td>3060</td>
<td>FF07FF16E2F819AC3</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>3070</td>
<td>025A5F7880A2325</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td>3080</td>
<td>025055617FD766</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>3090</td>
<td>05C3F2E227027C</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>3100</td>
<td>0E558E132997B</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>3110</td>
<td>BFB9463B2250586D</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>3120</td>
<td>01982A174F33PC4</td>
<td>274</td>
<td></td>
</tr>
<tr>
<td>3130</td>
<td>0FF57C943A311F</td>
<td>275</td>
<td></td>
</tr>
<tr>
<td>3140</td>
<td>07528B09F838ED</td>
<td>276</td>
<td></td>
</tr>
<tr>
<td>3150</td>
<td>05C5F8C6485338E</td>
<td>277</td>
<td></td>
</tr>
<tr>
<td>3160</td>
<td>0000F617FD766</td>
<td>279</td>
<td></td>
</tr>
</tbody>
</table>

#### ADDITIONAL TABLES

| DATA1 | ; SINE 7.0 X |
| DATA2 | ; SINE 7.5 X |

#### MC00F55617FD66

<table>
<thead>
<tr>
<th>IN</th>
<th>3200</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>-23170, -1608, 20787, 22412, 27245, 7962, -15446, -30652</td>
</tr>
<tr>
<td>DEC</td>
<td>-30273,-14010, 9512, 28105, 32137, 19519, -3212,-24279</td>
</tr>
<tr>
<td>DEC</td>
<td>-30273,-24279,-3212, 19519, 32137, 28105, 9512,-14010</td>
</tr>
<tr>
<td>DEC</td>
<td>-30273,-20862,-14446, 7962, 27245, 3212, 20787, -1608</td>
</tr>
<tr>
<td>DEC</td>
<td>-12539,-29621,-31356,-16845, 6393, 26319, 32609, 22005</td>
</tr>
<tr>
<td>DEC</td>
<td>0, 22005, 32609, 26319, 6393,-16846,-31356,-29621</td>
</tr>
</tbody>
</table>

#### DATA2

<table>
<thead>
<tr>
<th>IN</th>
<th>3800</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>-8855,-20942,-22169,-11910, 4520, 18670, 25927, -30273</td>
</tr>
<tr>
<td>DEC</td>
<td>-16381,-1137, 14997, 22916, 19262, 5629,-10921,-21812</td>
</tr>
<tr>
<td>DEC</td>
<td>-21403,-9905, 6725, 19870, 22721, 13800,-2271,-17165</td>
</tr>
<tr>
<td>DEC</td>
<td>-23166,-17165,-2271, 13800, 22721, 19870, 6725,-9905</td>
</tr>
<tr>
<td>DEC</td>
<td>-21403,-21812,-10920, 5629, 19262, 22916, 14686,-1137</td>
</tr>
<tr>
<td>DEC</td>
<td>-16381,-21328,-17906,-3239, 12871, 22472, 20431, 7984</td>
</tr>
<tr>
<td>DEC</td>
<td>-8855,-20942,-22169,-11910, 4520, 18670, 25927, -30273</td>
</tr>
<tr>
<td>DEC</td>
<td>-16381,-1137, 14997, 22916, 19262, 5629,-10921,-21812</td>
</tr>
<tr>
<td>DEC</td>
<td>-21403,-9905, 6725, 19870, 22721, 13800,-2271,-17165</td>
</tr>
<tr>
<td>DEC</td>
<td>-23166,-17165,-2271, 13800, 22721, 19870, 6725,-9905</td>
</tr>
</tbody>
</table>

#### DATA2

<table>
<thead>
<tr>
<th>IN</th>
<th>3800</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>-15558, 23055, 18607, 4520,-11910,-22169,-20942</td>
</tr>
<tr>
<td>DEC</td>
<td>-8865, 7804, 20431, 22472, 12970,-3399,-17908,-23138</td>
</tr>
<tr>
<td>DEC</td>
<td>-16381,-1137, 14997, 22916, 19262, 5629,-10921,-21812</td>
</tr>
<tr>
<td>DEC</td>
<td>-21403,-9905, 6725, 19870, 22721, 13800,-2271,-17165</td>
</tr>
<tr>
<td>DEC</td>
<td>-23166,-17165,-2271, 13800, 22721, 19870, 6725,-9905</td>
</tr>
<tr>
<td>DEC</td>
<td>-21403,-21812,-10920, 5629, 19262, 22916, 14686,-1137</td>
</tr>
<tr>
<td>DEC</td>
<td>-16381,-21328,-17906,-3239, 12871, 22472, 20431, 7984</td>
</tr>
</tbody>
</table>

#### DATA2

<table>
<thead>
<tr>
<th>IN</th>
<th>3800</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>-15558, 23055, 18607, 4520,-11910,-22169,-20942</td>
</tr>
<tr>
<td>DEC</td>
<td>-8865, 7804, 20431, 22472, 12970,-3399,-17908,-23138</td>
</tr>
<tr>
<td>DEC</td>
<td>-16381,-1137, 14997, 22916, 19262, 5629,-10921,-21812</td>
</tr>
<tr>
<td>DEC</td>
<td>-21403,-9905, 6725, 19870, 22721, 13800,-2271,-17165</td>
</tr>
<tr>
<td>DEC</td>
<td>-23166,-17165,-2271, 13800, 22721, 19870, 6725,-9905</td>
</tr>
</tbody>
</table>
MCS-96 MACRO ASSEMBLER     FFT_MAIN_APNOTE
02/18/86 PAGE 8

<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F00</td>
<td>0000C241C35E2148</td>
<td>302</td>
<td>DCW 0, 16834, 24259, 16465, 3188, -13029, -21886, -10657</td>
</tr>
<tr>
<td>3F11</td>
<td>0581E4343A443654</td>
<td>304</td>
<td>DCW -7842, 7384, 19011, 21553, 13425, -1958, -17103, 23821</td>
</tr>
<tr>
<td>3F20</td>
<td>088A5F88B532B45F</td>
<td>305</td>
<td>DCW -17629, -1819, 15601, 24356, 19816, 4710, -12341, 22232</td>
</tr>
<tr>
<td>3F30</td>
<td>068B8D65F1B3F49</td>
<td>306</td>
<td>DCW -20379, -6519, 7007, 18791, 21383, 13656. -1067, 15888</td>
</tr>
<tr>
<td>3F40</td>
<td>02565F8765527468</td>
<td>307</td>
<td>DCW -23166, -18442, -3475, 15942, 24059, 20959, 6442, -1230</td>
</tr>
<tr>
<td>3F50</td>
<td>065A7D4E4DE4A419</td>
<td>308</td>
<td>DCW -22427, -21392, -9500, 6548, 18708, 21475, 13892, -454</td>
</tr>
<tr>
<td>3F60</td>
<td>0B645A96B65B1800</td>
<td>309</td>
<td>DCW -14933, -22456, -18712, -6940, 12017, 23391, 21851, 6225</td>
</tr>
<tr>
<td>3F70</td>
<td>05D9C8A84A55865</td>
<td>310</td>
<td>DCW -5889, -22586, -22451, -10791, 5367, 18745, 21851, 14281</td>
</tr>
<tr>
<td>3F80</td>
<td>0111111111111111</td>
<td>311</td>
<td>END</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.
Listing 3—Main Routine (Continued)
Several constants are then setup for other routines. The purpose of centrally locating these constants was the ease of modifying the operation of the routines. Note that AVR_NUM and SCALE_FACTOR must be changed at the same time. SCALE_FACTOR is the shift count used to divide each FFT output value before it is added to the output array. AVR_NUM must be less than 2**SCALE_FACTOR or an overflow could occur. Next, the public variables are declared for the arrays and a few other parameters.

The program then begins by setting the stack pointer and waiting for the SBE-96 to finish talking to the terminal. If this is not done, there may be serial port interrupts occurring for the first twenty five milliseconds of program operation.

Initialization of the plotter is next, followed by setting the FFT_MODE byte. This byte controls the graphing, loading and magnitude calculation of the FFT data. Since FFT_MODE is declared PUBLIC in this module, and EXTERNAL in the PLOT module and FFTRUN module, the extra bits available in this byte can be used for future enhancements.

The next step is to clear the FFT output array. Since the FFT program can be set to average its results by dividing the output before adding it to the magnitude array, the array must be cleared before beginning the program.

Data is then loaded into the FFT input array by the code at LOAD_DATA, or the code at TABLE_LOAD, depending on the value of FFT_MODE bit 0. The tabled data located at DATA0 is a square wave of magnitude 1. This waveform provides a reasonable test of the FFT algorithm, as many harmonics are generated. The results are also easy to check as the pattern contains half zeros, imaginary values which are always the same, and real values which decrease. Figure 13 shows the output in fractions, hexadecimal and decimal. The hexadecimal and decimal values are based on an output of 16384 being equal to 1.00.

Note that the magnitude is

\[ \text{SQR (REAL}^2 + \text{IMAG}^2) \]

and the dB value is

\[ 10 \text{ LOG ( (REAL}^2 + \text{IMAG}^2)/65536 ) \]

The divide by 65536 is used for the dB scale to provide a reasonable range for calculations. If this was not done, a 32-bit LOG function would have been needed.

After the data is loaded, the data is optionally windowed, based on FFT_MODE bit 1, and the FFT program is called. Once the loop has been performed AVR_CNT times, the graph is drawn by the plot routine.

Appended to the main routine is the FFTOUT.M96 Listing. This is provided by the relocator and linker, RL96. With this listing and the main program, it is possible to determine which sections of code are at which addresses.

Using the modular programming methods employed here, it is reasonably easy to debug code. By emulating the program in a relatively high level language, each routine can be checked for functionality against a known standard. The closer the high level implementation matches the ASM96 version, the more possible checkpoints there are between the two routines.

Once all of the program routines (modules) can be shown to work individually, the main program should work unless there is unwanted interaction between the modules. These interactions can be checked by verifying the inputs and outputs of each module. The assembly language locations to perform the program breaks can be retrieved by absolutely locating the main module. The other modules can be dynamically located by RL96.

The more interactive program modules are, the more difficult the program becomes to debug. This is especially true when multiple interrupts are occurring, and several of the interrupt routines are themselves interruptable. In these cases, it may be necessary to use debugging equipment with trace capability, like the VLSICE-96. If this type of equipment is not available, then using I/O ports to indicate the entering and leaving of each routine may be useful. In this way it will be possible to watch the action of the program on an oscilloscope or logic analyzer. There are several places within this code that I/O port toggling has been used as an aid to debugging the program. These lines of code are marked "FOR INDICATION ONLY."
<table>
<thead>
<tr>
<th>K</th>
<th>Fractional</th>
<th>dB</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>REAL</td>
<td>IMAG</td>
<td>MAG²</td>
<td>REAL</td>
</tr>
<tr>
<td>0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1</td>
<td>0.0625</td>
<td>-1.2722</td>
<td>1.2738</td>
<td>38.225</td>
</tr>
<tr>
<td>2</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>3</td>
<td>0.0625</td>
<td>-0.4213</td>
<td>0.4260</td>
<td>28.710</td>
</tr>
<tr>
<td>4</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>5</td>
<td>0.0625</td>
<td>-0.2495</td>
<td>0.2572</td>
<td>24.329</td>
</tr>
<tr>
<td>6</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>7</td>
<td>0.0625</td>
<td>-0.1747</td>
<td>0.1855</td>
<td>21.491</td>
</tr>
<tr>
<td>8</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>9</td>
<td>0.0625</td>
<td>-0.1321</td>
<td>0.1462</td>
<td>19.421</td>
</tr>
<tr>
<td>10</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>11</td>
<td>0.0625</td>
<td>-0.1043</td>
<td>0.1216</td>
<td>17.820</td>
</tr>
<tr>
<td>12</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>13</td>
<td>0.0625</td>
<td>-0.0843</td>
<td>0.1049</td>
<td>16.540</td>
</tr>
<tr>
<td>14</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>15</td>
<td>0.0625</td>
<td>-0.0690</td>
<td>0.0931</td>
<td>15.499</td>
</tr>
<tr>
<td>16</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>17</td>
<td>0.0625</td>
<td>-0.0566</td>
<td>0.0844</td>
<td>14.645</td>
</tr>
<tr>
<td>18</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>19</td>
<td>0.0625</td>
<td>-0.0464</td>
<td>0.0778</td>
<td>13.944</td>
</tr>
<tr>
<td>20</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>21</td>
<td>0.0625</td>
<td>-0.0375</td>
<td>0.0729</td>
<td>13.374</td>
</tr>
<tr>
<td>22</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>23</td>
<td>0.0625</td>
<td>-0.0296</td>
<td>0.0691</td>
<td>12.918</td>
</tr>
<tr>
<td>24</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>25</td>
<td>0.0625</td>
<td>-0.0224</td>
<td>0.0664</td>
<td>12.564</td>
</tr>
<tr>
<td>26</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>27</td>
<td>0.0625</td>
<td>-0.0157</td>
<td>0.0644</td>
<td>12.305</td>
</tr>
<tr>
<td>28</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>29</td>
<td>0.0625</td>
<td>-0.0093</td>
<td>0.0632</td>
<td>12.135</td>
</tr>
<tr>
<td>30</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>31</td>
<td>0.0625</td>
<td>-0.0031</td>
<td>0.0626</td>
<td>12.051</td>
</tr>
</tbody>
</table>

Figure 13. FFT Output for a Square Wave Input

9.0 ANALOG TO DIGITAL CONVERTER MODULE

The module presented in Listing 4 is a general purpose one which converts analog values under interrupt control and stores them in one of two buffers. These buffers can then be downloaded to another buffer, such as the input buffer to the FFT program. During downloading, this module can convert the data into signed or unsigned formats, and fill a linear or a paired array. A paired array is like the one used in the FFT transform program. It requires N data points placed alternately in two arrays, one starting at zero and the other at N/2.
A2D_Buffering_Utility

This utility fills a memory buffer with A/D conversion results. The conversions are done under interrupt control, and are initiated when A2D_BUFF_Util is called. The results of the conversions are placed in one of two buffers, called BUFF0 and BUFF1.

This utility provides options for the selection of the buffer lengths, data format, sample period, conversion channel and time base. The utility also has a download routine that will load either buffer into a register file.

The data can be formatted as signed or unsigned linear or paired arrays.

RUN-TIME OPTIONS

Rather than use the STACK to pass controls, this utility gets its directions from 2 control words in memory. The utility expects that its control words are valid at the time A2D_BUFF_Util is called and remain valid throughout A/D interrupt executions and downloads. The control words are:

- Control_A2D ; BYTE ; Control information for the utility:
  - BIT#
  - 0-2 ; Channel Number
  - 3 ; Signed Result/Unsigned Result#
  - 4 ; Convert/Downloads#
  - 5 ; BUFF1/BUFFO# for conversions
  - BUFF0/BUFF1# for downloads
  - 6 ; Linear/Paried#
  - 7 ; Converter BUSY/IDLE#
The following is a table of equates that can be used to simplify the 
bit daddling requirements. If you are not running conversions concurrently
with downloads, always LSR Control_A2D with the following command then
ORB Control_A2D with the channel number you wish to convert if you are
starting a conversion.

Once the utility is called, care must be taken when Control_A2D is
modified. You can cause downloads to occur while conversions are running,
but you cannot start conversions during a download. To do this, ORB to the
control byte with the appropriate bits set. Do NOT change the BUFF bit or
the BUSY bit. Just set the download bit and set the data format bits to the
correct values.

The BUFF bit has opposite definitions for conversions and downloads. This
allows conversions to be done into BUFF0 while downloads come from BUFF1, and
vice versa.

A2D Utility Commands

c0n_b0  equ  00000000b; convert to BUFF0
c0n_b1  equ  00110000b; " BUFF1
dump_b0_l_u  equ  01100000b; download BUFF0 as LINEAR UNSIGNED data
dump_b1_l_u  equ  01000000b; " BUFF1 " " " "
dump_b0_p_u  equ  00100000b; " BUFF0 " PAIRED " "
dump_b1_p_u  equ  00000000b; " BUFF1 " " " 
dump_b0_l_s  equ  01010000b; download BUFF0 as LINEAR SIGNED data
dump_b1_l_s  equ  01001000b; " BUFF1 " " " 
dump_b0_p_s  equ  00101000b; " BUFF0 " PAIRED " "
dump_b1_p_s  equ  00010000b; " BUFF1 " " " 
$eject
### MMC-96 MACRO ASSMBLY

#### A2D_BUFFERING.Utility

**ERR LOC** | **OBJECT** | **LINE** | **SOURCE STATEMENT**
---|---|---|---
75 | | | :
76 | | | : ASSEMBLY-TIME OPTIONS
77 | | | :
78 | | | : The base addresses and length of each conversion buffer and the destination
79 | | | : buffer are DECLARED EXTERN in this utility. Other options such as selection
80 | | | : of the timer used as a timebase, the length of the buffer, and the effective
81 | | | : number of bits in the reported result are set at assembly time through use
82 | | | : of EQUates in this module.
83 | | | :
84 | | | : The following parameters need to be provided at assembly or link time.
85 | | | : The buffer bases are declared EXTERN by this utility, while the buffer
86 | | | : length shift count and ESO commands are EQUated.
87 | | | :
88 | | | : BUFFO_BASE : The starting address of BUFFO
89 | | | : BUFFI_BASE : The starting address of BUFFI
90 | | | : DEST_BUFF_BASE : The starting address of the download
91 | | | : : target buffer.
92 | | | :
93 | | | : BUFF_LENGTH : The number of SAMPLES that each
94 | | | : : buffer must hold must be >1 and <256
95 | | | :
96 | | | : Shift_count : The number of times that the conversion result is
97 | | | : : to be shifted right from its natural left justified
98 | | | : : position. Setting a shift count greater than 6 will
99 | | | : : result in lost bits to the right. Rounding is NOT
100 | | | : : done.
101 | | | :
102 | | | : CLOCK : Specify as either TIMER1 or T2CLK. This is the
103 | | | : : timebase used for conversions.
104 | | | :
105 | | | : Samples are stored as words in the buffers. The program stores
106 | | | : conversions linearly in BUFFO and BUFFI, and linearly or paired in the
107 | | | : destination buffer as selected. If the download is to be paired, the first
108 | | | : sample is placed in location DEST_BUFF_BASE, the second sample is placed in
109 | | | : location (DEST_BUFF_BASE + BUFF_LENGTH), the third in (DEST_BUFF_BASE + 2),
110 | | | : the fourth in (DEST_BUFF_BASE + 2 + BUFF_LENGTH), etc.
111 | | | :
112 | | | : select

---

**02/18/86 PAGE 3**
When a utility call directs the initiation of a set of A2D conversions, the first conversion is begun at approximately one sample time plus 50 state times from when the utility was called. This assumes that no interrupts are present.

The conversion busy bit is set approximately 50 state times after a call to the utility, if the convert bit was set in the A2D_Control byte. The busy bit is cleared after all conversion results have been stored in the result buffer designated (BUFF0 or BUFF1). Take great care in modifying the A2D_Control byte to do a download while conversions are taking place. You can never download a buffer that is being converted into. The results would be invalid.
MCS-96 MACRO ASSEMBLER  A2D_BUFFERING.Utility

ERR LOC OBJECT  LINE  SOURCE STATEMENT
0000  130  
0001  131  RESEG
0002  132  
0003  133  EINT4_BUF0_BASE, BUF1_BASE, DEST_BUF_BASE
0004  134  EINT4 ad_command, ad_result_lo, ad_result_hi
0005  135  EINT4 hso_command, hso_time,sp
0006  136  
0040  137  BUFI_LENGTH  EQU  64
0040  138  Shift_Count  EQU  1
0040  139  CLOCK  EQU  TIMER1
0040  140  
0040  141  ; set up hso commands for correct timer  ******************************
0040  142  
0040  143  TIMER1  equ  0AH
0040  144  TICLK  equ  0CH
0040  145  
0040  146  MASK  equ  (10h+CLOCK)AND(40h)
0040  147  
0040  148  Start_A2D  equ  (00001111b)OR(MASK)
0040  149  ; start a2d based on timer 1, no interrupt
0040  150  
0040  151  HSO_0_Low  equ  (00000000b)OR(MASK)
0040  152  ; make hso.0 low based on timer1 no interrupt
0040  153  
0040  154  HSO_0_High  equ  (00100000b)OR(MASK)
0040  155  ; make hso.0 hi based on timer1 no interrupt
0040  156  
0040  157  ; set up storage  ***************************************
0040  158  
0040  159  
0040  160  Control_A2D:  DSW  1; the byte that controls the utility execution
0040  161  
0040  162  Control_temp0:  DSW  1; temp register for download calls
0040  163  
0040  164  Control_temp1:  DSW  1; temp registers for conversion calls
0040  165  
0040  166  top_of_buffer:  DSW  1
0040  167  sample_count:  DSB  1
0040  168  DForm  equ  3 ; Signed/Unsigned#
0040  169  Convert/Download#
0040  170  Buffl/Buff0# for conversions
0040  171  Buff0/Buff1# for downloads
0040  172  Linear/Paried#
0040  173  Busy  equ  1000000B ; Bit 8
0040  174  $eject

270189-58
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000A</td>
<td></td>
<td>176</td>
<td>Sample_Period: DSW 1; the word that specifies the number of clock ticks that elapse between each sample</td>
</tr>
<tr>
<td></td>
<td></td>
<td>178</td>
<td>PUBLIC Control_A2D, Sample_Period</td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td>0000</td>
<td>src_ptr: DSW 1; some overlayable temp registers</td>
</tr>
<tr>
<td>0002</td>
<td></td>
<td>0004</td>
<td>dest_ptr: DSW 1</td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td>0004</td>
<td>loop_count: DSW 1</td>
</tr>
<tr>
<td>2002</td>
<td>AC00</td>
<td>2002</td>
<td>DOW A2D_DONE_Vector</td>
</tr>
<tr>
<td>0000</td>
<td></td>
<td>0000</td>
<td>CSEG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>PUBLIC A2D_BUFF_Util</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>Load_HSO_Command MACRO var ; Macro to load HSO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>LD hso_command,#var</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>LD hso_time,wdctemp0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>$@ject</td>
</tr>
<tr>
<td>Line</td>
<td>Source Statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>A2D_BUFF_Utils:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MCS-96 MACRO ASSEMBLER

A2D_BUFFERINGUTILITY 02/18/86 PAGE 8

ERR LOC OBJECT  LINE  SOURCE STATEMENT
0041 A1400004  R 254  ; Convert the data in the destination buffer
0045 A1000000  R 255  LD loop_count,#BUFF_LENGTH
0045 A1000000  R 255  LD src_ptr,#DEST BUFF_BASE
0049 A200000  R 257  Again: LD adu temp0,[src_ptr]
0040 7100000  R 258  ANDB adu temp0,#11000000b
004F 3300000  R 259  JBC Control_A2D, BFormat, Unsigned_Result

0052 004115  R 261  Convert_Data:
0052 69007700  R 262  SUB adu temp0,#7fe0H
0056 040100  R 263  SEHA adu temp0,#Shift_Count
0059 2003  R 264  BR Replace_Sample
005B 2066  R 265  Unsigned_Result:
005B 080100  R 267  SHR adu temp0, #Shift_Count

005E 2069  R 268  Replace_Sample:
005E C20000  R 270  ST adu temp0,[src_ptr]+
0061 B00105  R 271  DJNZ loop_count,Again ; Loop until done
0064 F0  R 272  RET

0065 276  R 273  Convert: ; Prepare to Start Conversions
0065 F2  R 274  PUSHF

0066 910059  R 275  ODR Control_A2D, #Busy ; set converter busy bit
0069 B13F08  R 281  LDB sample_count,#BUFF_LENGTH - 1
006C A1000006  R 282  LD top_of_buffer,#BUFF0_BASE
0070 A1800004  R 284  LD adu temp1,#(BUFF0_BASE + 2#BUFF_LENGTH)
0074 350098  R 285  JBC Control_A2D, B0_B1, Start_Conversions
0077 A1000000  R 287  LD top_of_buffer,#BUFF1_BASE
007B A1800004  R 289  $eject

270189-61
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>007F</td>
<td></td>
<td>290</td>
<td>Start_Conversions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>291</td>
<td>Start_Conversions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>292</td>
<td>Load channel number</td>
</tr>
<tr>
<td>007F</td>
<td>51070900</td>
<td>293</td>
<td>ANDB ad_command,Control_A2D,#00000111b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>294</td>
<td>; start first conversion</td>
</tr>
<tr>
<td>0083</td>
<td>446A0A02</td>
<td>295</td>
<td>ADD aductemp0,CLOCK,Sample_Period</td>
</tr>
<tr>
<td></td>
<td></td>
<td>296</td>
<td>; sample time from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>297</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>298</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>299</td>
<td>Load_HSO_Command Start_A2D</td>
</tr>
<tr>
<td>008D</td>
<td>CDD0</td>
<td>300</td>
<td>Start A2D at Time=aductemp0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>301</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>302</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>303</td>
<td>POP temp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>304</td>
<td>; get a copy of the paw</td>
</tr>
<tr>
<td></td>
<td></td>
<td>305</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>306</td>
<td>Load_HSO_Command HSO_0_high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>307</td>
<td>; set hso.0 high at conversion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>308</td>
<td>; start time for external S/H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>309</td>
<td></td>
</tr>
<tr>
<td>0096</td>
<td>81020200</td>
<td>310</td>
<td>OR temp,#202h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>311</td>
<td>; enable a2d interrupts</td>
</tr>
<tr>
<td>0099</td>
<td>646A0A02</td>
<td>312</td>
<td>ADD aductemp0,Sample_Period</td>
</tr>
<tr>
<td></td>
<td></td>
<td>313</td>
<td>; start second conversion one</td>
</tr>
<tr>
<td></td>
<td></td>
<td>314</td>
<td>; sample time from the first</td>
</tr>
<tr>
<td></td>
<td></td>
<td>315</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>316</td>
<td>Load_HSO_Command Start_A2D</td>
</tr>
<tr>
<td>00A2</td>
<td>CB00</td>
<td>317</td>
<td>; start second conversion one</td>
</tr>
<tr>
<td></td>
<td></td>
<td>318</td>
<td>; sample time from the first</td>
</tr>
<tr>
<td></td>
<td></td>
<td>319</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>320</td>
<td>Load_HSO_Command HSO_0_low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>321</td>
<td>; lower hso.0 for external S/H</td>
</tr>
<tr>
<td></td>
<td></td>
<td>322</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>323</td>
<td>PUSH temp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>324</td>
<td>; put paw back on stack</td>
</tr>
<tr>
<td>00AA</td>
<td>F3</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>00AB</td>
<td>FO</td>
<td>326</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>327</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>328</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>329</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>330</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>331</td>
<td>eject</td>
</tr>
</tbody>
</table>
Listing 4-A to D Converter Routine (Continued)

ASSMIBLY COMPLETED, NO ERROR(S) FOUND.
The listing contains a fairly complete description of what the program does. The block by block operations are shown below:

Lines 1-198 describe the program, declare the variables and set up equates. Several of these variables are declared as overlayable, so the user needs to be careful if using this module for other than the FFT program.

Lines 205-210 declare a macro which is used to load the HSO unit. This will be used repeatedly through the code.

Lines 212-253 determine whether a conversion or download has been requested. If a download has been requested, the data is downloaded to the destination array as either paired or linear data. Paired data has been described earlier.

Lines 255-278 contain a subroutine which converts the destination array to either signed or unsigned numbers. The numbers are also shifted right to provide the desired full-scale value as requested by SHIFT_COUNT.

Lines 279-334 describe the conversion routine. HSO.O is toggled with the start of each routine so that an external sample and hold can be used. The instructions in lines 308, 316, and 326 have been interweaved with the Load_HSO_Commands to provide the required 8 state delays between HSO loadings. If this was not done, NOPs would have been needed. It is easier to understand the code if these lines are thought of as being gathered at line 326.

Lines 337-353 are the actual A/D interrupt routine. The A/D results are placed BYTE by BYTE on the buffer, the A/D is reloaded, and then the number of samples taken is compared to the number needed. Note that the A/D command register needs to be reloaded even if the channel does not change. INCB on line 348 is used to insure that the DJNZ falls through on the next pass (if sample_count is not reset).

Lines 355-396 complete the routine. The HSO is set up to trigger the next conversion and provide the HSO.0 toggle for an external sample and hold. Once again, the time between consecutive loads of the HSO is 8 states minimum. Note that this section of code has been optimized for speed by reducing branches to an absolute minimum and duplicating code where needed.

This concludes the description of the A to D buffer module. In the FFT program, this module is run, then the FFT transform module, then the plot module. This allows variables to be overlaid, saving RAM space. The time cost for this is not bad, considering the printer is the limiting factor in these conversions. If more RAM was provided, and the FFT was run with its data in external RAM, this module could be run simultaneously with the other modules.

10.0 DATA PLOTTING MODULE

The plot module is relatively straight-forward, and is shown in Listing 5. After the declarations, which include overlayable registers, an initialization routine is listed. This separately called routine sets up the serial port on the 8096 to talk to the printer. In this case, the port has to be set for 300 baud.

A console out routine follows. This routine can also be called by any program, but it is used only by the plot routine in this example. The write to port 1 is used to trace the program flow. The character to be output is passed to this routine on the stack. This conforms to PLM-96 requirements.

Since all stack operations on the 8096 are 16-bits wide, a multiple character feature has been added to the console out routine. If the high byte it receives is non-zero, the ASCII character in that byte is printed after the character in the low byte. If the high byte has a value between 128 and 255, the character in the low byte is repeated the number of times indicated by the least significant 7 bits of the high byte.

The print decimal number routine is next. It is called with two words on the stack. The first word is the unsigned value to be printed. The second byte contains information on the number of places to be printed and zero and blank suppression. This routine is not overflow-proof. The user must declare a sufficient number of places to be printed for all possible numbers.

The DRAW_GRAPH routine provides the plot. It first sends a series of carriage return, line feeds (CRLFs) to clear the printer and provides a margin on the paper. Each row is started with the row number, 2 spaces, and a "+". Asterisks are then plotted until

Number of asterisks > FFT Value / PLOT.RES

Recall that PLOT.RES is a variable set by the main program. When the number of asterisks hits the desired value, the value of the line is printed. If the Decibel mode is selected, the line value is divided by 512 and printed in integer + decimal part form, followed by "dB". If the number of asterisks reaches PLOT_MAX, no value is printed. The next line is then started. A line with only a "!" is printed before the next plot line to provide a more aesthetic display on the printer. If a CRT was used, this extra line would probably not be wanted.
This program produces a plot on serially connected printer. The magnitude of each of the 32 input values is plotted horizontally, with one "!" followed by a line feed between each plot line. Each plot line starts with a "+" and the entire plot begins with 3 line feeds and ends with a form feed. The values to be plotted are 32 unsigned words based at the externally defined pointer PLOT_IN.

The routine INIT_OUTPUT must be run to set up the serial port when the system is turned on. CON_OUT can be used by a program to output to the serial port. DRAW_GRAPH is the routine that automatically plots the data.

Sizing of the graph can be done using PLOT_RES, which determines how many units are needed for each dot, and PLOT_MAX, which is the maximum value the program will be passed. Note that (PLOT_MAX/PLOT_RES) defines the maximum number of columns the routine will print.

**SOURCE STATEMENT**

```plaintext
1 $pagelen(50)
2
3 PLOT_SERIAL MODULE STACKSIZE (6)

; Intel Corporation, December 12, 1985
; by Ira Horden, MO Applications

; This program produces a plot on serially connected printer. The
; magnitude of each of the 32 input values is plotted horizontally, with one
; ";!" followed by a line feed between each plot line. Each plot line starts
; with a "+" and the entire plot begins with 3 line feeds and ends with a form
; feed. The values to be plotted are 32 unsigned words based at the externally
; defined pointer PLOT_IN.

; The routine INIT_OUTPUT must be run to set up the serial port when the
; system is turned on. CON_OUT can be used by a program to output to the
; serial port. DRAW_GRAPH is the routine that automatically plots the data.

; Sizing of the graph can be done using PLOT_RES, which determines how many
; units are needed for each dot, and PLOT_MAX, which is the maximum value the
; program will be passed. Note that (PLOT_MAX/PLOT_RES) defines the maximum
; number of columns the routine will print.

0000 EXTERN ioc1, baud_reg, spcom, spstat, abuf, port1
0000 EXTERN zero, ax, bx, cx, dx, FFT_MODE
0000 aptmp: db 1

0024 ORG 24H
0024 value: dsl 1
0028 divisor: dsl 1
002c xptr: daw 1
0030 yptr: daw 1
0034 xval: daw 1
0038 log_val: daw 1

0000 DSEG
0000 EXTERN PLOT_IN

0000 $}ject
```
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td></td>
<td>42</td>
<td>CSEG at 2500H</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>44</td>
<td>PUBLIC INIT_OUTPUT, COM_OUT, DRAW_GRAPH</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>46</td>
<td>EXTRN PLOT_RES, PLOT_RES_2, PLOT_MAX</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>48</td>
<td>INIT_OUTPUT:</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>50</td>
<td>; INITIALIZE SERIAL PORT</td>
</tr>
<tr>
<td>2500</td>
<td>B12000</td>
<td>51</td>
<td>ldb io1, #00100000B ; set p2.0 to txd</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>52</td>
<td>baud_val equ 624 ; 624=300 baud (at 12 MHz)</td>
</tr>
<tr>
<td>0002</td>
<td></td>
<td>53</td>
<td>baud_high equ ((baud_val-1)/256) OR 00H ; set for XTAL1 clock</td>
</tr>
<tr>
<td>000F</td>
<td></td>
<td>55</td>
<td>baud_low equ (baud_val-1) MOD 256</td>
</tr>
<tr>
<td>2503</td>
<td>B16F00</td>
<td>57</td>
<td>ldb baud_reg,#baud_low</td>
</tr>
<tr>
<td>2506</td>
<td>B18200</td>
<td>58</td>
<td>ldb baud_reg,#baud_high</td>
</tr>
<tr>
<td>2509</td>
<td>B1A900</td>
<td>59</td>
<td>ldb scom,#01010010b ; enable receiver mode 1</td>
</tr>
<tr>
<td>250C</td>
<td>B12000</td>
<td>61</td>
<td>ldb scom,#00100000b ; set Ti-temp</td>
</tr>
<tr>
<td>250F</td>
<td>FO</td>
<td>62</td>
<td>RET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>$ject</td>
</tr>
</tbody>
</table>

02/18/86
CONSOLE OUT ROUTINE

Call with a word parameter on stack. The low byte has the character

the character will be sent. If the high byte has a value between 61H and 8FH, the
to be sent. If the high byte has a value between 61H and 8FH, the character
character is repeated 1 to 126 times respectively. One repeat means
character will be printed 2 times. If the high byte contains
that the character will be printed 2 times. If the high byte contains
a value between 1 and 7FH, the character represented by that value will
a value between 1 and 7FH, the character represented by that value will
be printed after the character in the low byte. If the high byte
be printed after the character in the low byte. If the high byte
contains a value of zero only the low byte will be printed.
contains a value of zero only the low byte will be printed.

CON_OUT:

pop ax
: cx contains the calling address

pop dx

jbs dx+1,7,onechr
: If bit 7 is set print one character

cmpb dx+1,zero

je onechr
: if highbyte=0 print one character

twochr:

orb sptap,spstat
: wait for TI

jbc sptap,5,twochr

andb sptap,#10111111b
: clear TI-tap

orb zero,spstat
: remove possible false TI

IOB sbuf,dx
Load second character

ldi dx,de+1
: clear count byte

cld

andb dx,#07FH
: mask M6

onechr:

incb dx+1

andb dx,#07FH

wait1:
 orb sptap,spstat
: wait for TI

jbc sptap,5,wait1

andb sptap,#10111111b
: clear TI-tap

orb zero,spstat
: remove possible false TI

ldi sbuf,dx

DJNZ dx+1,wait1
: Effectively a RET

BR [ax] Effectively a RET
PRINT DECIMAL NUMBER ROUTINE

Call with two words on stack. The first is the value to be printed.

The second has mode information in the low byte.

MODE: 000 = suppress all zeros
001 = print all numbers
010 = suppress all zeros except rightmost
011 = do not print leading blanks

The high byte of the 2nd word = 2x the number of places to be printed

254C
254C C000 E 121  pop cx
254C C000 E 122  pop bx
2550 AC0100 E 123  lodsb  dx,bx+1
2553 A309962528 E 124  ld divisor,divtab[dx]
2558 CC24 125  pop value
256A 126  div_loop:
256A 127  clr value+2
256C 9C28E4 128  div value,divisor ; divide ax, dx by divisor
256F 380017 E 129  jba bx,0,chr_ok ; print character regardless of value
2562 980024 E 130  cmpb value,zero
2565 D70F 131  jne non_0 ; jump if value is non zero
2567 132  Val_O: ; Value is zero
2567 133  jbe jba ; Print space instead of 0
2567 134  protap: jba bx,1,protap ; Print space instead of 0
2567 135  prntsp: jba bx,2,cont ; Do not print space if bit is set
2570 A1F00024 136  ld value,#OF0H ; OF0h:30h = 20h = space
2574 2003 137  br chr_ok
2576 910100 E 138  non_0: orb bx,#0001B ; Set flag so 0's will be printed
2579 65300024 140  chr_ok: add value,#30h ; 30h+n = 0 to 9 ascii
257D 817F0024 E 141  end value,#7Fh ; send least sig seven bits, clear upper word
2581 C204 142  push value
2583 2F88 143  call con_out ; output ascii result (result=9)
2585 A20224 E 144  cont: ld value,value+2 ; load value with remainder
2588 D21A 145  clr divisor+2
258A 8D00A028 146  divu divisor,#10 ; next lower power of ten
258E 980028 E 147  cmp divisor,zero
2591 DC7C 148  jne div_loop
2593 149  div_done:
2593 E300 E 150  br [cx]
2596 152  DIVTAB: ; Number of places for result
2596 0000010000A006400 153  daw 0, 1, 10, 100, 1000, 10000 ; divisor table - 10##n
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>25A2</td>
<td>DRAW_GRAPE;</td>
<td>158</td>
<td>; Graph drawing routine</td>
</tr>
<tr>
<td>25A2</td>
<td>C90000</td>
<td>159</td>
<td>push #0dh</td>
</tr>
<tr>
<td>25A2</td>
<td>2F69</td>
<td>160</td>
<td>call con_out</td>
</tr>
<tr>
<td>25A7</td>
<td>C90000</td>
<td>161</td>
<td>push #024AH</td>
</tr>
<tr>
<td>25A7</td>
<td>2F64</td>
<td>162</td>
<td>call con_OUt</td>
</tr>
<tr>
<td>25AC</td>
<td>C90000</td>
<td>163</td>
<td>push #00</td>
</tr>
<tr>
<td>25AF</td>
<td>2F5F</td>
<td>164</td>
<td>call con_out</td>
</tr>
<tr>
<td>25B1</td>
<td>012C</td>
<td>165</td>
<td>clr xptr</td>
</tr>
<tr>
<td>25B3</td>
<td>0130</td>
<td>166</td>
<td>clr xval</td>
</tr>
<tr>
<td>25B5</td>
<td>G90000</td>
<td>167</td>
<td>NXT_BOW:</td>
</tr>
<tr>
<td>25B5</td>
<td>G90000A</td>
<td>168</td>
<td>push #0A0DH</td>
</tr>
<tr>
<td>25B6</td>
<td>2F66</td>
<td>169</td>
<td>call con_OUt</td>
</tr>
<tr>
<td>25B6</td>
<td>C90000</td>
<td>170</td>
<td>push #0DH</td>
</tr>
<tr>
<td>25B7</td>
<td>2F51</td>
<td>171</td>
<td>call con_OUt</td>
</tr>
<tr>
<td>25BF</td>
<td>CB30</td>
<td>172</td>
<td>push xval</td>
</tr>
<tr>
<td>25C1</td>
<td>C90200</td>
<td>173</td>
<td>push #0A0DH or 0010b</td>
</tr>
<tr>
<td>25C4</td>
<td>2F86</td>
<td>174</td>
<td>call PRINT_NUM</td>
</tr>
<tr>
<td>25C6</td>
<td>C92020</td>
<td>175</td>
<td>push #2020H</td>
</tr>
<tr>
<td>25C9</td>
<td>2F45</td>
<td>176</td>
<td>call con_OUt</td>
</tr>
<tr>
<td>25CB</td>
<td>CB5000</td>
<td>177</td>
<td>push #2NH</td>
</tr>
<tr>
<td>25CE</td>
<td>2F40</td>
<td>178</td>
<td>call con_out</td>
</tr>
<tr>
<td>25DD</td>
<td>A100002E</td>
<td>179</td>
<td>ld yptr,#PLOT_RES_2</td>
</tr>
<tr>
<td>25D4</td>
<td>180</td>
<td>181</td>
<td>; PLOT_RES2 = PLOT_RES/2</td>
</tr>
<tr>
<td>25D4</td>
<td>182</td>
<td>183</td>
<td>; PLOT_RES is defined 7 lines down</td>
</tr>
<tr>
<td>25D4</td>
<td>184</td>
<td>185</td>
<td>; Next Column</td>
</tr>
<tr>
<td>25D4</td>
<td>186</td>
<td>187</td>
<td>cmp yptr,PLOT_IN[xptr]</td>
</tr>
<tr>
<td>25D4</td>
<td>188</td>
<td>189</td>
<td>jn PRNT_NUM</td>
</tr>
<tr>
<td>25D9</td>
<td>D911</td>
<td>190</td>
<td>PRT_ME:</td>
</tr>
<tr>
<td>25D9</td>
<td>191</td>
<td>192</td>
<td>push #2AH</td>
</tr>
<tr>
<td>25D9</td>
<td>193</td>
<td>194</td>
<td>call con_out</td>
</tr>
<tr>
<td>25D9</td>
<td>195</td>
<td>196</td>
<td>INC_CNT:</td>
</tr>
<tr>
<td>25D9</td>
<td>197</td>
<td>198</td>
<td>add yptr,#PLOT_RES</td>
</tr>
<tr>
<td>25D9</td>
<td>199</td>
<td>200</td>
<td>; PLOT_RES = number of inputs per output point</td>
</tr>
<tr>
<td>25D9</td>
<td>201</td>
<td>202</td>
<td>cmp yptr,#PLOT_MAX</td>
</tr>
<tr>
<td>25D9</td>
<td>203</td>
<td>204</td>
<td>; PLOT_MAX = maximum line length</td>
</tr>
<tr>
<td>25D9</td>
<td>205</td>
<td>206</td>
<td>jn nat_col</td>
</tr>
<tr>
<td>25D9</td>
<td>207</td>
<td>208</td>
<td>br NXTLN</td>
</tr>
<tr>
<td>25D9</td>
<td>209</td>
<td>210</td>
<td>$eject</td>
</tr>
</tbody>
</table>

02/18/86 PAGE 5
<table>
<thead>
<tr>
<th>MCS-96 MACRO ASSEMBLER</th>
<th>PLOT_SERIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRK LOC</td>
<td>OBJECT</td>
</tr>
<tr>
<td>258C</td>
<td>259C 8900002E</td>
</tr>
<tr>
<td>2590</td>
<td>2590 C0F9</td>
</tr>
<tr>
<td>2592</td>
<td>2592 C52020</td>
</tr>
<tr>
<td>2595</td>
<td>2595 3F19</td>
</tr>
<tr>
<td>2597</td>
<td>2597 8B00B</td>
</tr>
<tr>
<td>259A</td>
<td>259A CB20000</td>
</tr>
<tr>
<td>259C</td>
<td>259C C9000A</td>
</tr>
<tr>
<td>2601</td>
<td>2601 2F49</td>
</tr>
<tr>
<td>2603</td>
<td>2603 2035</td>
</tr>
<tr>
<td>2605</td>
<td>2605 A3200002E</td>
</tr>
<tr>
<td>2609</td>
<td>2609 A2F90</td>
</tr>
<tr>
<td>2610</td>
<td>2610 C800</td>
</tr>
<tr>
<td>2612</td>
<td>2612 C9002A</td>
</tr>
<tr>
<td>2615</td>
<td>2615 2F35</td>
</tr>
<tr>
<td>2617</td>
<td>2617 C92800</td>
</tr>
<tr>
<td>261A</td>
<td>261A 2F4</td>
</tr>
<tr>
<td>261C</td>
<td>261C 802801</td>
</tr>
<tr>
<td>261F</td>
<td>261F 1100</td>
</tr>
<tr>
<td>2621</td>
<td>2621 D8200</td>
</tr>
<tr>
<td>2625</td>
<td>2625 370102</td>
</tr>
<tr>
<td>2628</td>
<td>2628 0700</td>
</tr>
<tr>
<td>262A</td>
<td>262A C800</td>
</tr>
<tr>
<td>263C</td>
<td>263C C90016</td>
</tr>
<tr>
<td>263F</td>
<td>263F 3F18</td>
</tr>
<tr>
<td>2641</td>
<td>2641 C92000</td>
</tr>
<tr>
<td>2644</td>
<td>2644 2F84</td>
</tr>
<tr>
<td>2646</td>
<td>2646 C96442</td>
</tr>
<tr>
<td>2649</td>
<td>2649 2E05</td>
</tr>
</tbody>
</table>

**SOURCE STATEMENT**

<table>
<thead>
<tr>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>198</td>
<td>PNT_NUM:</td>
</tr>
<tr>
<td>200</td>
<td>cmp yptr,#PLOT_RES_2 ; If value is less than minimum needed</td>
</tr>
<tr>
<td>201</td>
<td>be NXTLN ; for a plot, do not print value</td>
</tr>
<tr>
<td>202</td>
<td>push #20200H ; print 2 spaces then value</td>
</tr>
<tr>
<td>204</td>
<td>call con_out</td>
</tr>
<tr>
<td>205</td>
<td>JBS FFT_MODE,3,db_mode</td>
</tr>
<tr>
<td>207</td>
<td>norm_mode:</td>
</tr>
<tr>
<td>208</td>
<td>push PLOT_IN[xptr]</td>
</tr>
<tr>
<td>209</td>
<td>push #0A00H or 0000B ; suppress all zeros</td>
</tr>
<tr>
<td>210</td>
<td>call PRINT_NUM</td>
</tr>
<tr>
<td>211</td>
<td>BR NXTLN</td>
</tr>
<tr>
<td>212</td>
<td>push ax</td>
</tr>
<tr>
<td>214</td>
<td>ld yptr,plot_in[xptr] ; PLOT_IN = 512*10^LOG(x)</td>
</tr>
<tr>
<td>215</td>
<td>shr yptr,#1 ; yptr=265 * 10^LOG(x)</td>
</tr>
<tr>
<td>216</td>
<td>ldb xe,ax,yptr+1 ; ax=10^LOG(x) = yptr/256</td>
</tr>
<tr>
<td>217</td>
<td>push ax ; Print AX</td>
</tr>
<tr>
<td>218</td>
<td>push ax ; Print AX</td>
</tr>
<tr>
<td>219</td>
<td>push #0A00H or 0010B ; supress all but rightmost zero</td>
</tr>
<tr>
<td>220</td>
<td>call PRINT_NUM</td>
</tr>
<tr>
<td>221</td>
<td>push #2EH ; Decimal point</td>
</tr>
<tr>
<td>222</td>
<td>call con_out</td>
</tr>
<tr>
<td>223</td>
<td>db_mode:</td>
</tr>
<tr>
<td>224</td>
<td>ldb ax+1,yptr ; high byte of ax = fractional portion of</td>
</tr>
<tr>
<td>225</td>
<td>cl rb ax ;</td>
</tr>
<tr>
<td>226</td>
<td>10LOG(x)</td>
</tr>
<tr>
<td>227</td>
<td>m ul x,.$368H ; if ax=F700H then ax+2 now = 998 decimal</td>
</tr>
<tr>
<td>228</td>
<td>jbc ax+1,7,no_rnd ; print all numbers to three places</td>
</tr>
<tr>
<td>229</td>
<td>inc dx ; round value up</td>
</tr>
<tr>
<td>230</td>
<td>no_rnd:</td>
</tr>
<tr>
<td>231</td>
<td>push dy ; dy=ax+2</td>
</tr>
<tr>
<td>232</td>
<td>push #600H or 0010H ; print all numbers to three places</td>
</tr>
<tr>
<td>233</td>
<td>call Print_num</td>
</tr>
<tr>
<td>234</td>
<td>push #20H ; space</td>
</tr>
<tr>
<td>235</td>
<td>call con_out</td>
</tr>
<tr>
<td>236</td>
<td>push #4264H ; &quot;db&quot;</td>
</tr>
<tr>
<td>237</td>
<td>call con_out</td>
</tr>
</tbody>
</table>

270189-69
ASS_LBR PLOT_SERIAL

<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>263B</td>
<td>C900A</td>
<td>240</td>
<td>push #0AAH</td>
</tr>
<tr>
<td>263E</td>
<td>290</td>
<td>241</td>
<td>call CON_OUT</td>
</tr>
<tr>
<td>2640</td>
<td>C90000</td>
<td>242</td>
<td>push #0DEH</td>
</tr>
<tr>
<td>2643</td>
<td>29C</td>
<td>243</td>
<td>call CON_OUT</td>
</tr>
<tr>
<td>2645</td>
<td>C90086</td>
<td>244</td>
<td>push #600H</td>
</tr>
<tr>
<td>2648</td>
<td>29C</td>
<td>245</td>
<td>call CON_OUT</td>
</tr>
<tr>
<td>264A</td>
<td>C92100</td>
<td>246</td>
<td>push #21H</td>
</tr>
<tr>
<td>264D</td>
<td>29C1</td>
<td>247</td>
<td>call CON_OUT</td>
</tr>
<tr>
<td>264F</td>
<td>0730</td>
<td>248</td>
<td>inc xval</td>
</tr>
<tr>
<td>2651</td>
<td>650002C</td>
<td>249</td>
<td>add xptr,#2</td>
</tr>
<tr>
<td>2655</td>
<td>820002C</td>
<td>250</td>
<td>cmp xptr,#02</td>
</tr>
<tr>
<td>2659</td>
<td>02022756</td>
<td>251</td>
<td>ble nxt_row</td>
</tr>
<tr>
<td>265C</td>
<td>C900A</td>
<td>252</td>
<td>push #0AAH</td>
</tr>
<tr>
<td>2660</td>
<td>29AE</td>
<td>253</td>
<td>call CON_OUT</td>
</tr>
<tr>
<td>2662</td>
<td>C9000C</td>
<td>254</td>
<td>push #000H</td>
</tr>
<tr>
<td>2665</td>
<td>29A9</td>
<td>255</td>
<td>call CON_OUT</td>
</tr>
<tr>
<td>2667</td>
<td>F0</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>266B</td>
<td>END</td>
<td>257</td>
<td></td>
</tr>
</tbody>
</table>

; Setup for next line
; CR LF

; nul
; 7 spaces

; !

; Start printing next row

; CR LF ; Form feed for next graph

; null, FF

; ASSEMBLY COMPLETED, NO ERROR(S) FOUND.

02/18/86 PAGE 7

270189-70
At the end of the plot, a form feed is given to set the printer up for the next graph. Our printer would frequently miss the character after a CRLF. To solve this problem, a null (ASCII 0) is sent after every CRLF to make sure the printer is ready for the next line. This has been found to be a problem with many devices running at close to their maximum capacity, and the nulls work well to solve it.

With the plot completed, the program begins to run again by taking another set of A to D samples.

11.0 USING THE FFT PROGRAM

The program can be used with either real or tabled data. If real data is used, the signal is applied to analog channel 1. The program as written performs A/D samples at 100 microsecond intervals, collecting the 64 samples in 6.4 milliseconds. This sets the sampling window frequency at 156 Hz. If tabled data is used, 64 words of data should be placed in the location pointed to by DATA0 in the TABLE_LOAD routine of the Main Module.

Program control is specified by FFT_MODE which is loaded in the main module. Also within the main module are settings which control the A to D buffer routine and the Plot routine. The intention was to have only one module to change and recompile to vary parameters in the entire program.

The program modules are set up to run one-at-a-time so that the code would be easy to understand. Additionally, the Plot routine takes so long relative to the other sections, that it doesn't pay to try to overlap code sections. If this code were to be converted to run a process instead of print a graph, it might be worthwhile to run the FFT and the A/D routines at the same time.

If the goal of a modified program is to have the highest frequency sampling possible, it might be desirable to streamline the A/D section and run it without interruption. When the A to D routine was complete the FFT routine could be started. The reasoning behind this is that at the fastest A/D speeds the processor will be almost completely tied up processing the A/D information and storing it away. Using an interrupt based A/D routine would slow things down.

A set of programs which will perform a FFT has been presented in this application note. These programs are available from the INSITE users library as program CA-26. More importantly, dozens of programming examples have been made available, making it easier to get started with the 8096. Examples of how to use the hardware on the 8096 have already appeared in AP-248, “Using The 8096.” These two applications notes form a good base for the understanding of MCS-96 microcontroller based design.
Matrices are a convenient way to express groups of equations. Consider the complex discrete Fourier Transform in equation 9, with \( N = 4 \).

\[
Y_n = \sum_{k=0}^{3} X(k) W^{nk} \quad n = 0, 1, 2, 3
\]

This can be expanded to

\[
\begin{align*}
Y(0) &= X(0) W^0 + X(1) W^1 + X(2) W^2 + X(3) W^3 \\
Y(1) &= X(0) W^1 + X(1) W^0 + X(2) W^3 + X(3) W^2 \\
Y(2) &= X(0) W^2 + X(1) W^3 + X(2) W^0 + X(3) W^1 \\
Y(3) &= X(0) W^3 + X(1) W^2 + X(2) W^1 + X(3) W^0
\end{align*}
\]

In matrix notation, this is shown as

\[
\begin{bmatrix}
Y(0) \\
Y(1) \\
Y(2) \\
Y(3)
\end{bmatrix} =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & W^1 & W^2 & W^3 \\
1 & W^2 & W^0 & W^2 \\
1 & W^3 & W^2 & W^1
\end{bmatrix}
\begin{bmatrix}
X(0) \\
X(1) \\
X(2) \\
X(3)
\end{bmatrix}
\]

The first step to simplifying this is to reduce the center matrix. Recalling that

\[
W^N = W^{N \mod N} \quad \text{and} \quad W^0 = 1
\]

The matrix can be reduced to have less non-trivial multiplications.

\[
\begin{bmatrix}
Y(0) \\
Y(1) \\
Y(2) \\
Y(3)
\end{bmatrix} =
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & W^1 & W^2 & W^3 \\
1 & W^2 & W^0 & W^2 \\
1 & W^3 & W^2 & W^1
\end{bmatrix}
\begin{bmatrix}
X(0) \\
X(1) \\
X(2) \\
X(3)
\end{bmatrix}
\]

The square matrix can be factored into

\[
\begin{bmatrix}
Y(0) \\
Y(2) \\
Y(1) \\
Y(3)
\end{bmatrix} =
\begin{bmatrix}
1 & W^0 & 0 & 0 \\
1 & W^2 & 0 & 0 \\
0 & 0 & 1 & W^1 \\
0 & 0 & 1 & W^3
\end{bmatrix}
\begin{bmatrix}
1 & 0 & W^0 & 0 \\
0 & 1 & 0 & W^0 \\
1 & 0 & W^2 & 0 \\
1 & 0 & 1 & W^2
\end{bmatrix}
\begin{bmatrix}
X(0) \\
X(1) \\
X(2) \\
X(3)
\end{bmatrix}
\]

For this equation to work, the \( Y(1) \) and \( Y(2) \) terms need to be swapped, as shown above. This procedure is a Bit Reversal, as described in the text.

Multiplying the two rightmost matrices results in

\[
\begin{align*}
X(0) + X(2) W^0 \\
X(1) + X(3) W^0 \\
X(0) + X(2) W^3 \\
X(1) + X(3) W^2
\end{align*}
\]

requiring 4 complex multiplications & 4 complex additions

Noting that \( W^0 = -W^2 \), 2 of the complex multiplications can be eliminated, with the following results

\[
\begin{align*}
X(0) + X(2) W^0 \\
X(1) + X(3) W^0 \\
X(0) - X(2) W^0 \\
X(1) - X(3) W^0
\end{align*}
\]

since \( W^1 = -W^3 \), a similar result occurs when this vector is multiplied by the remaining square matrix. The resulting equations are:

\[
\begin{align*}
Y(0) &= (X(0) + X(2) W^0) + W^0 (X(0) + X(3) W^0) \\
Y(2) &= (X(0) + X(2) W^0) - W^0 (X(1) + X(3) W^0) \\
Y(1) &= (X(0) - X(2) W^0) + W^1 (X(1) - X(3) W^0) \\
Y(3) &= (X(0) - X(2) W^0) - W^1 (X(1) - X(3) W^0)
\end{align*}
\]

The number of complex multiplications required is 4, as compared with 16 for the unfactored matrix.

In general, the FFT requires

\[
\frac{N \cdot \text{EXponent}}{2} \quad \text{complex multiplications}
\]

and

\[
N \cdot \text{EXponent} \quad \text{complex additions}
\]

where

\[
\text{EXponent} = \log_2 N
\]

A standard Fourier Transform requires

\[
N^2 \quad \text{complex multiplications}
\]

and

\[
N(N-1) \quad \text{complex additions}
\]
The following plots are examples of output from the FFT program. These plots were generated using tabled data, but very similar plots have also been made using the analog input module. Typically, a plot made using the analog input module will not show quite as much power at each frequency and will show a positive value for the DC component. This is because it is difficult to get exactly a full-scale analog input with no DC offset.

Plot 1 is a Magnitude plot of a square wave of period NT.

Plot 2 is the same data plotted in dB. Note how the dB plot enhances the difference in the small signal values at the high frequencies.

Plot 3 shows the windowed version of this data. Note that the widening of the bins due to windowing shows energy in the even harmonics that is not actually present. For data of this type a different window other than Hanning would normally be used. Many window types are available, the selection of which can be determined by the type of data to be plotted.

Plot 4 shows a sine wave of period NT/7 or frequency 7/NT.

Plot 5 shows the same input with windowing. Note the signal shown in bins 6 and 8.

Plot 6 shows a sine wave of period NT/7.5. Note the noise caused by the discontinuity as discussed earlier.

Plot 7 uses windowing on the data used for plot 6. Note the cleaner appearance.

Plot 8 shows a sine wave input of magnitude 0.707 and period NT/7.5.

Plot 9 shows same input with windowing.

Plot 10 shows a sine wave of magnitude 0.707/16 and period NT/11.

Plot 11 shows the same input with windowing. Note that there is no power shown in bins 10 and 12. This is because at 6 dB down from 3 dB they are nearly equal to zero.

Plot 12 uses the sum of the signals for plots 8 and 10 as inputs. Note that the component at period NT/11 is almost hidden.

Plot 13 uses the same signal as plot 12 but applies windowing. Now the period component at NT/11 can easily be seen. The Hanning window works well in this case to separate the signal from the leakage. If the signals were closer together the Hanning window may not have worked and another window may have been needed.
Plot 1—Magnitude Plot of Squarewave
Plot 2—Decibel Plot of Squarewave

<table>
<thead>
<tr>
<th>Index</th>
<th>Decibel Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38.222 dB</td>
</tr>
<tr>
<td>1</td>
<td>28.706 dB</td>
</tr>
<tr>
<td>2</td>
<td>24.327 dB</td>
</tr>
<tr>
<td>3</td>
<td>21.487 dB</td>
</tr>
<tr>
<td>4</td>
<td>19.421 dB</td>
</tr>
<tr>
<td>5</td>
<td>17.815 dB</td>
</tr>
<tr>
<td>6</td>
<td>16.538 dB</td>
</tr>
<tr>
<td>7</td>
<td>15.499 dB</td>
</tr>
<tr>
<td>8</td>
<td>14.639 dB</td>
</tr>
<tr>
<td>9</td>
<td>13.940 dB</td>
</tr>
<tr>
<td>10</td>
<td>13.363 dB</td>
</tr>
<tr>
<td>11</td>
<td>12.908 dB</td>
</tr>
<tr>
<td>12</td>
<td>12.554 dB</td>
</tr>
<tr>
<td>13</td>
<td>12.296 dB</td>
</tr>
<tr>
<td>14</td>
<td>12.125 dB</td>
</tr>
<tr>
<td>15</td>
<td>12.043 dB</td>
</tr>
</tbody>
</table>

Plot of Decibel Levels for Squarewave Signal
<table>
<thead>
<tr>
<th></th>
<th>Decibels</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.105 dB</td>
</tr>
<tr>
<td>1</td>
<td>32.203 dB</td>
</tr>
<tr>
<td>2</td>
<td>28.678 dB</td>
</tr>
<tr>
<td>3</td>
<td>22.690 dB</td>
</tr>
<tr>
<td>4</td>
<td>20.760 dB</td>
</tr>
<tr>
<td>5</td>
<td>18.308 dB</td>
</tr>
<tr>
<td>6</td>
<td>16.990 dB</td>
</tr>
<tr>
<td>7</td>
<td>15.460 dB</td>
</tr>
<tr>
<td>8</td>
<td>14.476 dB</td>
</tr>
<tr>
<td>9</td>
<td>13.398 dB</td>
</tr>
<tr>
<td>10</td>
<td>12.620 dB</td>
</tr>
<tr>
<td>11</td>
<td>11.795 dB</td>
</tr>
<tr>
<td>12</td>
<td>11.175 dB</td>
</tr>
<tr>
<td>13</td>
<td>10.507 dB</td>
</tr>
<tr>
<td>14</td>
<td>10.000 dB</td>
</tr>
<tr>
<td>15</td>
<td>9.464 dB</td>
</tr>
<tr>
<td>16</td>
<td>9.039 dB</td>
</tr>
<tr>
<td>17</td>
<td>8.616 dB</td>
</tr>
<tr>
<td>18</td>
<td>8.281 dB</td>
</tr>
<tr>
<td>19</td>
<td>7.916 dB</td>
</tr>
<tr>
<td>20</td>
<td>7.628 dB</td>
</tr>
<tr>
<td>21</td>
<td>7.347 dB</td>
</tr>
<tr>
<td>22</td>
<td>7.121 dB</td>
</tr>
<tr>
<td>23</td>
<td>6.889 dB</td>
</tr>
<tr>
<td>24</td>
<td>6.706 dB</td>
</tr>
<tr>
<td>25</td>
<td>6.542 dB</td>
</tr>
<tr>
<td>26</td>
<td>6.409 dB</td>
</tr>
<tr>
<td>27</td>
<td>6.265 dB</td>
</tr>
<tr>
<td>28</td>
<td>6.191 dB</td>
</tr>
<tr>
<td>29</td>
<td>6.094 dB</td>
</tr>
<tr>
<td>30</td>
<td>6.082 dB</td>
</tr>
<tr>
<td>31</td>
<td>6.031 dB</td>
</tr>
</tbody>
</table>

Plot 3—Plot of Squarewave with Window
Plot 4—Sin (7.0X) without Window

36.121 dB
Plot 6—Sin (7.5X) without Window
Plot 7—Sin (7.5X) with Window
Plot 8—0.707 • Sin (7.5X) without Window
Plot 9—0.707 • Sin (7.5X) with Window
AP-275

Plot 10—0.707/16 · Sin (11X) without Window

9.031 dB
Plot 11—0.707/16 • Sin (11X) with Window
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>dB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>**********************************</td>
<td>11.242 dB</td>
</tr>
<tr>
<td>1</td>
<td>**********************************</td>
<td>11.425 dB</td>
</tr>
<tr>
<td>2</td>
<td>**********************************</td>
<td>11.936 dB</td>
</tr>
<tr>
<td>3</td>
<td>**********************************</td>
<td>12.846 dB</td>
</tr>
<tr>
<td>4</td>
<td>**********************************</td>
<td>14.296 dB</td>
</tr>
<tr>
<td>5</td>
<td>**********************************</td>
<td>16.561 dB</td>
</tr>
<tr>
<td>6</td>
<td>**********************************</td>
<td>20.409 dB</td>
</tr>
<tr>
<td>7</td>
<td>**********************************</td>
<td>29.425 dB</td>
</tr>
<tr>
<td>8</td>
<td>**********************************</td>
<td>28.959 dB</td>
</tr>
<tr>
<td>9</td>
<td>**********************************</td>
<td>19.000 dB</td>
</tr>
<tr>
<td>10</td>
<td>**********************************</td>
<td>14.187 dB</td>
</tr>
<tr>
<td>11</td>
<td>**********************************</td>
<td>13.105 dB</td>
</tr>
<tr>
<td>12</td>
<td>**********************************</td>
<td>8.472 dB</td>
</tr>
<tr>
<td>13</td>
<td>**********************************</td>
<td>6.483 dB</td>
</tr>
<tr>
<td>14</td>
<td>**********************************</td>
<td>4.819 dB</td>
</tr>
<tr>
<td>15</td>
<td>**********************************</td>
<td>3.382 dB</td>
</tr>
<tr>
<td>16</td>
<td>*****</td>
<td>2.152 dB</td>
</tr>
<tr>
<td>17</td>
<td>***</td>
<td>1.082 dB</td>
</tr>
</tbody>
</table>

Plot 12—0.707 (\(\sin(7.5X)\) + \(\frac{1}{16}\) \(\sin(11X)\)) without Window
Plot 13—0.707 (\sin (7.5X) + \frac{1}{16} \sin (11X)) with Window
BIBLIOGRAPHY

INTEL PUBLICATIONS
1. 1986 Microcontroller Handbook, Order Number 210918-004
2. Using the 8096, AP-248, Order Number 270061-001
3. MCS-96 Macro Assembler User’s Guide, Order Number 122048-001
4. MCS-96 Utilities User’s Guide, Order Number 122049-001
Upgrade Path from 8096-90 to 8096BH to 80C196
Converting applications that use an 8X9X-90 to use an 8X9XBH requires consideration of a few of the BH enhancements. Descriptions of each of the differences between the -90 and the BH follow, along with a discussion of the implications of the change.

BHE and INST are latched: The bus control signals BHE and INST are valid throughout the bus cycle on 8X9XBH devices. ON -90 devices, these signals need to be latched on the falling edge of ALE.

Byte Read following RESET rising: The bus control and buswidth options of 8X9XBH devices are selected by configuration of the chip immediately following the rising edge of RESET. During the usual 10 state reset sequence, BH parts will perform a byte read of location 2018H to acquire configuration information prior to fetching the first opcode at location 2080H. The 8X9X-90 does not perform this read.

ALE is high while in reset: The ALE/ADV pin of the 8X9XBH is driven high while the reset pin is held low. On -90 devices, ALE is driven low while in RESET. Circuits which rely on the state of ALE while reset is low must be modified. The reset state of ALE was changed to enable implementation of the Chip Configuration Byte read from external memory following the rising edge of RESET.

EA is latched on RESET rising: The 8X9XBH latches the value of EA on the rising edge of RESET. On -90 devices, EA was not latched and could be changed without placing the part in RESET. This change was necessary to enhance ROM/EPROM security. Circuits that rely on EA not being latched must be modified.

A/D speed increased: The 8X9X5BH and 8X97XH A/D converters complete conversion in 88 state times. On -90 devices with A/D converters, a conversion takes 168 state times. This translates in an increased conversion speed from 42μs on -90 parts to 22μs on BH parts running at 12MHz. Software that relies upon the speed of conversion for timing must be changed. It is also recommended that MCS-96 software be written so as to not be impacted by further changes in A/D conversion speed.

Sample/Hold on A/D: The 8X9X5BH and 8X97XH have a sample/hold on the input of the A/D converter. 8X9X-90 devices with A/D converters do not have sample/hold circuitry. External analog circuitry which also includes a sample/hold must provide a settled analog input within the first four state times of 8X9XBH conversion.

Duplicate Fetches: The 8X9XBH bus controller was made more aggressive when it comes to instruction fetches in order to minimize the execution speed degradation of using an 8-bit bus. As a result, instruction fetches over a 16-bit bus sometimes occur when there is no space in the prefetch queue to store the fetched opcodes. This requires another instruction fetch from the same address when space in the prefetch queue opens up.

To the external system, these occurrences appear as duplicate instruction fetches. An estimated 10 percent of all instruction fetches will be "duplicates", while overall bus loading will be approximately 65 to 70 percent, compared to an 8X9X-90 bus loading of approximately 55 to 60 percent. Execution speed is not impacted by a duplicate fetch.

Write Pulse Width: The 8X9XBH 16-bit bus write pulse width is one Tosc longer than on the 8X9X-90, thus allowing slower memories and peripherals to be used. In order to widen the WR pulse width, the time between the end of WR and the next ALE was reduced by Tosc. Note that the signals WRH, WR, and with an 8-bit bus are still the same width as on -90 parts.

Vpp Replaces Vbb: Vpp is the programming pin for EPROM devices. Systems that have this connected through a capacitor to ANGND (required on 8X9X-90 parts) do not need to change. ANGND must be held nominally at the same potential as VSS, and Vpp must NOT be connected to VCC. High voltage must NEVER be placed on the Vpp pin of a ROM device.

While there is almost no reason to do so, an application should not attempt to execute with the EA pin at logic zero and Vcc at 5.5 Vdc on an 879XBH EPROM device. Additionally, the design should always begin the "out of RESET" code execution from the internal EPROM, immediately after the power-on sequence.

Reserved location warning: Intel reserved addresses can not be used by applications which use 8X9X9BH internal ROM/EPROM. The data read from a reserved location is not guaranteed, and a write to any reserved location could cause unpredictable results. When attempting to program Intel Reserved addresses, the data must be OFFFH to ensure a harmless result.

Intel Reserved locations, when mapped to external memory, must be filled with OFFFH to ensure compatibility with future parts.

A positive transition on NMI: The 8X9XBH does not clear the Watchdog Timer. The 8X9X-90 does clear the WDT on a positive transition of NMI, and both part vector to external address 0000H.

The following is the latest information on upgrading a NMOS 8096 to a CHMOS 80C196.
The chip which is the CHMOS 8096BH replacement is designated the 80C196. The part can be configured to be pin compatible with the 8096, but because of the process change and other enhancements, it may not be plug compatible in some designs. This is to say that you will not be able to arbitrarily swap out a NMOS 8096 and replace it with the 80C196. However, if a few rules are followed the changes required will be almost painless.

80C196 OVERVIEW

First, some background on the 80C196 is needed. The opcode set is a true superset of the 8096, but some enhancements have been made to the peripherals and timings. The crystal is divided by 2 on the 80C196, instead of 3, as on the 8096. This means that the 80C196 running at 8 MHz will have a 250 ns state time, Just like an 8096 running at 12 MHz.

An 80C196 running at 8 MHz will emulate an 8096 at 12 MHz except that some of the instructions and peripherals will operate faster. Instructions which will be speeded up include mul, div, interrupt, call, ret, and jumps. The serial port will require a different baud value and the A to D may not run at exactly the same speed. This means that timing loops which measure instruction speed or A to D completion speed may have to be modified. The bus timings, while not nanosecond for nanosecond compatible, will work in most systems.

DESIGN GUIDELINES

1. Do not use undefined register areas for storage or depend on them to return a specific value if it is not stated in the Embedded Controller. Undefined registers and locations on this, or any other, part should be considered off-limits and reserved for development systems, testing or future use.

2. Do not base timings loops on instruction execution times, as some instructions may execute faster on the 80C196 than on the 8096, even when the 80C196 is slowed down to 8 MHz, its 8096 compatible rate. Counter-type loops should be initialized with values that can easily be changed at compile time.

3. Do not base critical timings on interrupt responses, A to D completions, flag settings, etc. This is for the same reason as above; some of these responses may be slightly different from those on the 8096. Timer 1 is provided for critical timings. With an 8 MHz crystal, it will increment every 2 microseconds, just as an 8096 running at 12 MHz.

4. The serial port baud register values should be easily changeable at compile time. Since the serial port is now capable of running at a higher frequency, a different baud rate value will be needed.

5. The circuitry interfacing to the chip should be capable of interfacing to the 80C196. The I/O lines on 80C196 will look a lot like those on the 80C51.

6. The BHE/WRH signal in eight bit and write strobe mode will go low for odd byte transfers and high for even byte transfers. The WR/WRL signal will go low for odd byte transfers and high for even byte transfers. Normally, the WR/WRL signal should go low for odd and even byte transfers since transfers are on the low byte of the data bus.

7. PUSH and POP operations addressed relative to the stack pointer work differently on the 80C196 than on the 8096. On the 8096, the address is calculated based on the un-updated stack pointer value, on the 80C196, the address is calculated based on the updated value. The only operations effected are: PUSH xx[sp], PUSH [sp], PUSH sp, POP xx[sp], POP [sp], POP sp.

8. The VPD pin on the 8X9X parts is now the CDE (Clock Detect Enable) pin on the 80C196. When tied high, CDE enables a clock speed sensor and will reset the part if the Xtal frequency drops below a few hundred KHz. While this is perfect for most production boards, it may be desirable to have a jumper option on this function for evaluation boards.
Memory Expansion for the 8096

DOUG YODER
ECO APPLICATIONS ENGINEER
This Application Brief presents two examples of a paging scheme for the 8096, allowing either 256K bytes of total memory, or 544K bytes of total memory. Both systems utilize \texttt{PORTI} as the output for the upper address lines. Because Interrupt vectors, and other critical sections of code must always be present, addresses 0-7FFFFH always refer to the same main page. The \texttt{PORTI} upper addresses only affect addresses 8000-FFFFFH, by slapping several 32K pages in and out.

**THE 256K SYSTEM**

**Hardware**

The hardware for the 256K system (see Figures 4 & 5, an example with 128K ROM and 128K RAM) utilizes a 74LS157 quad 2 to 1 multiplexer. The enable pin of the 74LS157 is tied to the inverted A15 signal, which is the latched addr/data 15 (ADI5) signal from the 96. In this way, when A15 is low, the 74LS157 is disabled and all its outputs are low. Particularly, MA17 is low, which selects the 27512 and deselects the rams. Also, MA15 and MA16 are low, which guarantee that addresses 0-7FFFH of the 27512 are accessed.

When A15 is high, the 74LS157 is enabled to pass MA15 - MA17 values. The bank select pin of the 74LS157 is connected to the INST pin of the 96. When the INST pin is high, for a code access, INSTA15 - INSTA17 (PORT1.0 - PORT1.2) are used. When INST is low, for a data read or write, DATAA15 - DATAA17 (PORT1.3 - PORT 1.5) are used. This allows for the use of separate pages for code and data without having to change the upper address lines each time. Also, it is possible to select a ROM page for a data table, or load a RAM page with executable code downloaded from another source. PORT1.6 and PORT1.7 can still be used as I/O ports. If a -90 part were used, the INST pin would need to be latched since it is only valid during the address output on the bus pins.

This system was designed to get the maximum amount of memory with a minimum amount of hardware. The amount of ROM and RAM was picked arbitrarily, and could be reconfigured in various ways, however, this may require slight modifications or additions to the decoder circuitry. This setup has a main page at addresses 0-7FFFFH, and upper pages 1-7 at addresses 8000-FFFFFH. Note that upper page 0 is the same as the main page. The WRL and WRH feature of the BH part was used to allow for byte writes to RAM. If the -90 part were to be used, additional logic would be necessary to generate these signals from WR and BHE.

The RAM chips utilized were NEC uPD43256-15 32K x 8 static rams with an access time of 150ns. The ROMs were Intel 27512 64K x 8 EPROMs with an access time of 200ns. The decoder circuitry used was entirely LS TTL. Using an 8097BH running at 10MHz, there was ample time for address decoding and memory access. Timing analysis showed that 12MHz operation would also be accommodated easily. If slower memories are used, further analysis would be necessary. Also, it would be possible to switch to 74128S to greatly decrease the decoding response time.

**Software**

When using this system there are several things to keep in mind when preparing the software.

Since ASM96 will only allow addresses from 0-FFFFFH, it is necessary to generate each page of code in a separate file. These pages should not be linked together, but rather should each be used to program the proper section of the EPROM associated with that page. The main page routine should be coded with addresses from 0-7FFFFH, and each of the upper pages should be coded with addresses from 8000-FFFFFH. Because linking is not possible, each module should contain a table of constants which defines the symbols used in other modules. These values are easily obtained from the listing file, which can be created using zeros in the table the first time. The addresses of the pages in a 27512 after splitting low and high bytes into 2 EPROMs are shown in Figure 1.

<table>
<thead>
<tr>
<th>EPROM LOCATION</th>
<th>EPROM LOCATION</th>
<th>RAM LOCATION</th>
<th>RAM LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATION U5</td>
<td>LOCATION U6</td>
<td>LOCATION U7</td>
<td>LOCATION U8</td>
</tr>
<tr>
<td>0H</td>
<td>0H</td>
<td>0H</td>
<td>0H</td>
</tr>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>PAGE4</td>
<td>PAGE4</td>
</tr>
<tr>
<td>PAGE</td>
<td>PAGE</td>
<td>LOW</td>
<td>HIGH</td>
</tr>
<tr>
<td>LOW</td>
<td>HIGH</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
<tr>
<td>3FFFH</td>
<td>3FFFH</td>
<td>3FFFH</td>
<td>3FFFH</td>
</tr>
<tr>
<td>PAGE1</td>
<td>PAGE1</td>
<td>PAGE5</td>
<td>PAGE5</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>BYTE8</td>
<td>BYTE8</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
<tr>
<td>4000H</td>
<td>4000H</td>
<td>4000H</td>
<td>4000H</td>
</tr>
<tr>
<td>PAGE2</td>
<td>PAGE2</td>
<td>PAGE6</td>
<td>PAGE6</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>BYTE8</td>
<td>BYTE8</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
<tr>
<td>7FFFFH</td>
<td>7FFFFH</td>
<td>7FFFFH</td>
<td>7FFFFH</td>
</tr>
<tr>
<td>PAGE3</td>
<td>PAGE3</td>
<td>PAGE7</td>
<td>PAGE7</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>BYTE8</td>
<td>BYTE8</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
<tr>
<td>8000H</td>
<td>8000H</td>
<td>8000H</td>
<td>8000H</td>
</tr>
<tr>
<td>PAGE4</td>
<td>PAGE4</td>
<td>PAGE4</td>
<td>PAGE4</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>BYTE8</td>
<td>BYTE8</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
<tr>
<td>BFFFH</td>
<td>BFFFH</td>
<td>BFFFH</td>
<td>BFFFH</td>
</tr>
<tr>
<td>PAGE5</td>
<td>PAGE5</td>
<td>PAGE5</td>
<td>PAGE5</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>BYTE8</td>
<td>BYTE8</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
<tr>
<td>C000H</td>
<td>C000H</td>
<td>C000H</td>
<td>C000H</td>
</tr>
<tr>
<td>PAGE6</td>
<td>PAGE6</td>
<td>PAGE6</td>
<td>PAGE6</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>BYTE8</td>
<td>BYTE8</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
<tr>
<td>FFFFH</td>
<td>FFFFH</td>
<td>FFFFH</td>
<td>FFFFH</td>
</tr>
<tr>
<td>PAGE7</td>
<td>PAGE7</td>
<td>PAGE7</td>
<td>PAGE7</td>
</tr>
<tr>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
<td>LOW</td>
</tr>
<tr>
<td>BYTE8</td>
<td>BYTE8</td>
<td>BYTES</td>
<td>BYTES</td>
</tr>
</tbody>
</table>

Figure 1. The Current System

6-182
All changes to the upper instruction addresses of PORT1 must be made by code located in the main page. A listing of subroutines for use in the main page, and a listing of macros for use in all pages is provided. By invoking one of these macros the programmer can easily transfer from one page to another, or select a new data page. The subroutines should not be called directly, they should be entered by using the appropriate macro. The subroutines should be located at the addresses specified, otherwise the macros must be changed as they are written to call an absolute address in the main page. Also, any hardware changes may render the software inoperative.

Because the WRL-WRH feature of the 96BH is used, the correct Chip Configuration Register value of OFBH must be loaded into the ROMs at address 2018H. This is done in the main code file with the following statements:

CSEG AT 2018H

CCR: DCB OFBH ;VALUE FOR CHIP CONFIGURATION REGISTER

Finally, it is necessary to initialize the DATA address at the start of the program this can be done using the NEW_DATA_ADDRESS MACRO.

THE 544K SYSTEM

Hardware

The hardware for the 544K system (see Figures 6 & 7, an example with 288K ROM and 256K RAM) has some slight changes from the 256K system.

First, all pins of PORT1 are now in use as address lines. This allows for PORT1 to select 16 pages of memory, with a different address for instructions or data.

Second, 27128 16K x 8 EPROMS have been added for use as the main code page. In this system, the main page is physically separate from upper page 0. The 27128's are selected by A15 being low. The upper pages of memory are selected when A15 is high which enables the 74LS155 demultiplexer which is used for address decoding. When the 74LS155 is disabled, its outputs are all high, which disables all upper memories. The 74LS157 is enabled all the time, to speed up address decoding, as its outputs do not matter when the 74LS155 is disabled.

Software

All rules for the 256K system apply to the 544K system, except that the main page no longer overlaps page 0. However, because all of PORT1 is now in use, different macros and subroutines must now be used. These have been included also.

THE INST PIN

The instruction pin has been verified to work correctly on the 8X9X-90, 8X9XBH, and the 80C196. The functionality of the INST pin is as follows.

Instruction Fetches

The INST pin is high during an external memory read indicating the read is an instruction fetch. This includes immediate data reads since the data is embedded in the code.

Data Reads and Writes

The INST is low during an external memory read or write indicating the bus cycle is a data cycle. This would be indirect and indexed instructions which are directed at external memory.
Figure 3. The 544K Memory Map
NOTE:
All other connections should be made as specified in current Embedded Controller Handbook.

Figure 4. 128K ROM + 128K RAM Memory
Figure 5. 128K ROM + 128K RAM Memory
;MACROS FOR 256K SYSTEM

;LONG_BRANCH IS INVOKED TO BRANCH FROM ONE PAGE TO ANOTHER.
;ADDRESS MUST HAVE A VALUE FROM 8000H TO FFFFH.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

LONG_BRANCH MACRO ADDRESS, NEW_PAGE
LD CODE_ADDRESS, #ADDRESS ;SET UP CODE_ADDRESS REGISTER
LDB NEW_PAGE_NO, NEW_PAGE ;SET UP NEW_PAGE_NO REGISTER
BR 7FFOH ;BRANCH TO I_P_BRANCH
ENDM

;LONG_CALL IS INVOKED TO CALL A SUBROUTINE IN ANOTHER PAGE.
;ADDRESS MUST HAVE A VALUE FROM 8000H TO FFFFH.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

LONG_CALL MACRO ADDRESS, NEW_PAGE
LD CODE_ADDRESS, #ADDRESS ;SET UP CODE_ADDRESS REGISTER
LDB NEW_PAGE_NO, NEW_PAGE ;SET UP NEW_PAGE_NO REGISTER
CALL 7FCOH ;CALL I_P_CALL
ENDM

;PUSH_OLD_DATAPAGE IS INVOKED TO INSTALL A NEW DATA PAGE AND SAVE
;THE OLD VALUE ON THE SYSTEM STACK.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

PUSH_OLD_DATAPAGE MACRO NEW_PAGE
LDB AL, PORT1 ;GET OLD PAGE NUMBER...
PUSH AX ;STORE IT ON THE STACK
LDB AL, NEW_PAGE ;GET NEW DATA PAGE NUMBER...
ANDB AL, #0000000B ;MASK IT...
SHLB AL, #3 ;SHIFT IT TO PROPER POSITION...
ANDB PORT1, #11000111B ;CLEAR THE OLD ONE...
ORB PORT1, AL ;AND LOAD IN NEW ONE
ENDM

;POP_OLD_DATAPAGE IS INVOKED TO REINSTALL AN OLD DATA PAGE THAT WAS SAVED
;ON THE SYSTEM STACK BY PUSH_OLD_DATAPAGE.

POP_OLD_DATAPAGE MACRO NEW_PAGE
POP AX ;RECALL OLD PAGE NUMBER...
ANDB AL, #00111000B ;MASK OLD ONE FOR DATA PAGE...
ANDB PORT1, #11000111B ;CLEAR NEW DATA PAGE...
ORB PORT1, AL ;AND LOAD IN OLD ONE
ENDM

;NEW_DATA_PAGE IS INVOKED TO INSTALL A NEW DATA PAGE.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

NEW_DATA_PAGE MACRO NEW_PAGE
LDB AL, NEW_PAGE ;GET NEW DATA PAGE NUMBER...
ANDB AL, #0000000B ;MASK IT...
SHLB AL, #3 ;SHIFT IT TO PROPER POSITION...
ANDB PORT1, #11000111B ;CLEAR THE OLD ONE...
ORB PORT1, AL ;AND LOAD IN NEW ONE
ENDM
; SUBROUTINES FOR 256K SYSTEM

CSEG AT 7FCOH

; SUBROUTINE: I_P_CALL
; THIS SUBROUTINE ALLOWS FOR THE CALLING OF SUBROUTINES LOCATED IN
; A DIFFERENT PAGE OF MEMORY.
; PARAMETERS: CODE_ADDRESS, NEW_PAGE_NO
; SUBROUTINES: ANY THAT ARE REQUESTED.
I_P_CALL:   LDB AL, PORT1  ; GET OLD PAGE NUMBER...
PUSH AX     ; STORE IT ON THE STACK
ANDB PORT1, #11111000B ; CLEAR OLD INST PAGE...
ANDB NEW_PAGE_NO, #00000111B ; MASK NEW ONE...
ORB PORT1, NEW_PAGE_NO ; AND LOAD IT IN
PUSH #I_P_RETURN ; SAVE RETURN ADDRESS...
BR [CODE_ADDRESS] ; CALL REQUESTED ROUTINE
I_P_RETURN: POP AX     ; RECALL OLD PAGE NUMBER...
ANDB PORT1, #11111000B ; CLEAR NEW INST PAGE...
ANDB AL, #00000111B ; MASK OLD ONE...
ORB PORT1, AL ; AND LOAD IT IN
RET ; RETURN TO CALLING ROUTINE

CSEG AT 7FFOH

; SUBROUTINE: I_P_BRANCH
; THIS SUBROUTINE ALLOWS FOR BRANCHING TO LOCATIONS IN A DIFFERENT
; PAGE OF MEMORY.
; PARAMETERS: CODE_ADDRESS, NEW_PAGE_NO
; SUBROUTINES: NONE
I_P_BRANCH:  ANDB PORT1, #11111000B ; CLEAR OLD INST PAGE...
ANDB NEW_PAGE_NO, #00000111B ; MASK NEW ONE...
ORB PORT1, NEW_PAGE_NO ; AND LOAD IT IN
BR [CODE_ADDRESS] ; BRANCH TO REQUESTED ROUTINE
NOTE:
All other connections should be made as specified in current Embedded Controller Handbook.

Figure 6. 288K ROM + 256K RAM Memory
Figure 7. 288K ROM + 256K RAM Memory
;MACROS FOR 544K SYSTEM

;LONG_BRANCH IS INVOKED TO BRANCH FROM ONE PAGE TO ANOTHER.
;ADDRESS MUST HAVE A VALUE FROM 8000H TO FFFFH.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

LONG_BRANCH MACRO ADDRESS, NEW_PAGE
LD CODE_ADDRESS, #ADDRESS ;SET UP CODE_ADDRESS REGISTER
LDB NEW_PAGE_NO, NEW_PAGE ;SET UP NEW_PAGE_NO REGISTER
BR 7FFOH ;BRANCH TO I_P_BRANCH
ENDM

;LONG_CALL IS INVOKED TO CALL A SUBROUTINE IN ANOTHER PAGE.
;ADDRESS MUST HAVE A VALUE FROM 8000H TO FFFFH.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

LONG_CALL MACRO ADDRESS, NEW_PAGE
LD CODE_ADDRESS, #ADDRESS ;SET UP CODE_ADDRESS REGISTER
LDB NEW_PAGE_NO, NEW_PAGE ;SET UP NEW_PAGE_NO REGISTER
CALL 7FCOH ;CALL I_P_CALL
ENDM

;PUSH_OLD_DATAPAGE IS INVOKED TO INSTALL A NEW DATA PAGE AND SAVE THE OLD
;VALUE ON THE SYSTEM STACK.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

PUSH_OLD_DAPAG MACRO NEW_PAGE
LDB AL, PORT1 ;GET OLD PAGE NUMBER...
PUSH AX ;STORE IT ON THE STACK
LDB AL, NEW_PAGE ;GET NEW DATA PAGE NUMBER...
SHLB AL, #4 ;SHIFT IT TO PROPER POSITION...
ANDB PORT1, #00001111B ;CLEAR THE OLD ONE...
ORB PORT1, AL ;AND LOAD IN NEW ONE
ENDM

;POP_OLD_DATAPAGE IS INVOKED TO REINSTALL AN OLD DATA PAGE THAT WAS SAVED
;ON THE SYSTEM STACK BY PUSH_OLD_DATAPAGE.

POP_OLD_DAPAG MACRO AX
POP AX ;RECALL OLD PAGE NUMBER...
ANDB AX, #11110000B ;MASK OLD ONE FOR DATA PAGE...
ANDB PORT1, #00001111B ;CLEAR NEW DATA PAGE...
ORB PORT1, AL ;AND LOAD IN OLD ONE
ENDM

;NEW_DATA_PAGE IS INVOKED TO INSTALL A NEW DATA PAGE.
;NEW_PAGE CAN BE AN IMMEDIATE NUMBER OR A REGISTER NUMBER.

NEW_DATA_PAGE MACRO NEW_PAGE
LDB AL, NEW_PAGE ;GET NEW DATA PAGE NUMBER...
SHLB AL, #4 ;SHIFT IT TO PROPER POSITION...
ANDB PORT1, #00001111B ;CLEAR THE OLD ONE...
ORB PORT1, AL ;AND LOAD IN NEW ONE
ENDM
;SUBROUTINES FOR 544K SYSTEM

CSEG AT 7FCOH

;SUBROUTINE:  I_P_CALL
:  THIS SUBROUTINE ALLLOWS FOR THE CALLING OF SUBROUTINES LOCATED IN
:  A DIFFERENT PAGE OF MEMORY.
:  
:  PARAMETERS:  CODE_ADDRESS, NEW_PAGE_NO
:  SUBROUTINES:  ANY THAT ARE REQUESTED.
:  
I_P_CALL:
  LDB  AL, PORT1 ;GET OLD PAGE NUMBER...
PUSH AX ;STORE IT ON THE STACK
ANDB PORT1, #11110000B ;CLEAR OLD INST PAGE...
ANDB NEW_PAGE_NO, #00001111B ;MASK NEW ONE...
ORB PORT1, NEW_PAGE_NO ;AND LOAD IT IN
PUSH #I_P_RETURN ;SAVE RETURN ADDRESS...
BR [CODE_ADDRESS] ;CALL REQUESTED ROUTINE

I_P_RETURN:
  POP AX ;RECALL OLD PAGE NUMBER...
ANDB PORT1, #11110000B ;CLEAR NEW INST PAGE...
ANDB AL, #00001111B ;MASK OLD ONE...
ORB PORT1, AL ;AND LOAD IT IN
RET ;RETURN TO CALLING ROUTINE

CSEG AT 7FFOH

;SUBROUTINE:  I_P_BRANCH
:  THIS SUBROUTINE ALLLOWS FOR BRANCHING TO LOCATIONS IN A DIFFERENT
:  PAGE OF MEMORY.
:  
:  PARAMETERS:  CODE_ADDRESS, NEW_PAGE_NO
:  SUBROUTINES:  NONE
:  
I_P_BRANCH:
  ANDB PORT1, #11110000B ;CLEAR OLD INST PAGE...
ANDB NEW_PAGE_NO, #00001111B ;MASK NEW ONE...
ORB PORT1, NEW_PAGE_NO ;AND LOAD IT IN
BR [CODE_ADDRESS] ;BRANCH TO REQUESTED ROUTINE
Integer Square Root Routine for the 8096

LIONEL SMITH
ECO APPLICATIONS ENGINEER
This Application Brief presents an example of calculating the square root of a 32-bit signed integer.

**Theory**

Newton showed that the square root can be calculated by repeating the approximation:

\[ X_{\text{new}} = \frac{(R/X_{\text{old}} + X_{\text{old}})}{2} \]

where:
- \( R \) is the radicand
- \( X_{\text{old}} \) is the current approximation of the square root
- \( X_{\text{new}} \) is the new approximation

until you get an answer you like. A common technique for deciding whether or not you like the answer is to loop on the approximation until \( X_{\text{new}} \) stops changing. If you are dealing with real (floating point) numbers this technique can sometimes get you in trouble because it's possible to hang up in the loop with \( X_{\text{new}} \) alternating between two values. This is not the case with integers. As an example of how it all works, consider taking the square root of 37 with an initial guess (\( X_{\text{old}} \)) of 1:

\[
\begin{align*}
X_{\text{new}} &= (37/1 + 1)/2 = 19; \quad X_{\text{old}} = 19 \\
X_{\text{new}} &= (37/19 + 19)/2 = 10; \quad X_{\text{old}} = 10 \\
X_{\text{new}} &= (37/10 + 10)/2 = 6; \quad X_{\text{old}} = 6 \\
X_{\text{new}} &= (37/6 + 6)/2 = 6; \quad X_{\text{old}} = 6 - \text{done!}
\end{align*}
\]

Note that in integer arithmetic the remainder of a division is ignored and the square root of a number is floored (i.e. the square root is the largest integer which, when multiplied by itself, is less than or equal to the radicand).

**Practice**

The only significant problem in implementing the square root calculation using this algorithm is that the division of \( R \) by \( X_{\text{old}} \) could easily be a 32 by 32 divide if \( R \) is a 32 bit integer. This is ok if you happen to have a 32 by 32 divide instruction, but most 16-bit machines (including the 8096) only provide a 32 by 16 divide. However, a little bit of creative laziness will allow us to squeeze by using the 32 by 16 bit divide on the 8096.

The largest positive integer you can represent with a 32-bit two's complement number is \( 07ff\text{ffffh} \), or \( 2,147,483,647 \). The square root of this number is \( 0b504h \), or 46,340. The largest square root that we can generate from a 32-bit radicand can be represented in 16-bits. If we are careful in picking our initial \( X_{\text{old}} \) we can do all of the divisions with the 32 by 16 divide instruction we have available. Picking the largest possible 16-bit number (\( 0fffh \)) will always work although it may slow the calculation down by requiring too many iterations to arrive at the correct result. The algorithm below takes a slightly more intelligent approach. It uses the normalize instruction to figure out how many leading zeros the 32-bit radicand has and picks an initial \( X_{\text{old}} \) based on this information. If there are 16 or more leading zeros then the radicand is less than 16 bits so an initial \( X_{\text{old}} \) of \( 0fh \) is chosen. If the radicand is more than 16 bits then the initial \( X_{\text{old}} \) is calculated by shifting the value \( 0fh \) by half as many places as there were leading zeros in the radicand. To give credit where credit is due, I first saw this "trick" in the January 1986 issue of Dr. Dobbs's Journal in a letter from Michael Barr of McGill University.

The routine was timed in a 12.0 MHz 8096 as it calculated the square roots of all positive 32-bit numbers, the following numbers include the CALL and return sequence and were measured using Timer 1 of the 8096.

- Minimum Execution Time: 24 microseconds
- Maximum Execution Time: 236 microseconds
- Average Execution Time: 102 microseconds

**Comments**

The program module which follows is part of a collection of routines which perform integer and real arithmetic on a software implemented tagged stack. The top element of the stack is call TOS and is in fixed locations in the register file. Since the square root operation only involves TOS, further details of the stack structure are not shown.
**Source File:** ROOT2.A96

**Object File:** ROOT2.OBJ

**Invocation Command Controls:** NOSB

**Line Source Statement:**

1: sqrt module
2: ; 32 bit integer square root for the 8096
3: ; public qstk_isqrt  ; TOP -> SQUARE_ROOT(TOP)
4: extrn interr:entry  ; Integer error routine
5: ; id stags for stack integer routines
6: isqrt_id equ 19h
7: ; error codes
8: overflow equ 00h
9: paramerr equ 01h
10: invalid_input equ 02h
11: oseg at 1ch
12: equ 18h :word
13: qstk_reg: word pointer to next element in the argument stack.
14: tos_tag equ (qstk_reg+2):word
15: tos_value:
16: dsl 1 ; 32 bit integer
17: oseg at 30h
18: ; ==========
19: qstk_reg:
20: dsl 1 ; make sure of alignment
21: next equ qstk_reg:word
22: tos_tag equ (qstk_reg+2):word
23: tos_value:
24: dsl 1 ; 32 bit integer
25: oseg
26: ; ======
27: b1 macro param
28: bnc param
29: endm
30: bhe macro param
31: bc param
32: endm
33: $eject
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>48</td>
<td>cseg</td>
</tr>
<tr>
<td>0000</td>
<td>49</td>
<td>SEG : ===&lt; ===</td>
</tr>
<tr>
<td>0000</td>
<td>50</td>
<td>; Top of argument stack</td>
</tr>
<tr>
<td>0000</td>
<td>51</td>
<td>qstk_isqrt:</td>
</tr>
<tr>
<td>0000</td>
<td>52</td>
<td>Takes the square root of the long integer in TOS</td>
</tr>
<tr>
<td>0000</td>
<td>53</td>
<td>; TOS-&gt; Top of argument stack</td>
</tr>
<tr>
<td>0000</td>
<td>54</td>
<td>iTOS - isqrt(TOS)</td>
</tr>
<tr>
<td>0020</td>
<td>55</td>
<td>Xold set cx</td>
</tr>
<tr>
<td>0000</td>
<td>56</td>
<td>ld ax,tos_value</td>
</tr>
<tr>
<td>0003</td>
<td>57</td>
<td>ld dx,(tos_value+2)</td>
</tr>
<tr>
<td>0006</td>
<td>58</td>
<td>be (dx+1),7,qst105 ; if (TOS &lt; 0)</td>
</tr>
<tr>
<td>0009</td>
<td>59</td>
<td>push #(isqrt_id*256+paramerr)</td>
</tr>
<tr>
<td>000C</td>
<td>60</td>
<td>call qstk_isqrtO</td>
</tr>
<tr>
<td>000F</td>
<td>61</td>
<td>ret ; Exit</td>
</tr>
<tr>
<td>0010</td>
<td>62</td>
<td>normal ax,bx</td>
</tr>
<tr>
<td>0013</td>
<td>63</td>
<td>cmp bx,#16</td>
</tr>
<tr>
<td>001A</td>
<td>64</td>
<td>cmp bx,#offh</td>
</tr>
<tr>
<td>001E</td>
<td>65</td>
<td>br qstk_isqrtO</td>
</tr>
<tr>
<td>0020</td>
<td>66</td>
<td>qst105;</td>
</tr>
<tr>
<td>0025</td>
<td>67</td>
<td>shr bx,#1</td>
</tr>
<tr>
<td>0027</td>
<td>68</td>
<td>ld Xold, #offfh</td>
</tr>
<tr>
<td>002A</td>
<td>69</td>
<td>clr bx xold= (ax+Xold)/2</td>
</tr>
<tr>
<td>003F</td>
<td>70</td>
<td>add Xold,ax</td>
</tr>
<tr>
<td>0042</td>
<td>71</td>
<td>addc bx,0</td>
</tr>
<tr>
<td>0045</td>
<td>72</td>
<td>shr Xold,#1</td>
</tr>
<tr>
<td>0048</td>
<td>73</td>
<td>br qstk_isqrtO</td>
</tr>
<tr>
<td>0050</td>
<td>74</td>
<td>do</td>
</tr>
<tr>
<td>005D</td>
<td>75</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>0060</td>
<td>76</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>0065</td>
<td>77</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>0072</td>
<td>78</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>0078</td>
<td>79</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>0084</td>
<td>80</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>008A</td>
<td>81</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>0090</td>
<td>82</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>0096</td>
<td>83</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00A2</td>
<td>84</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00A8</td>
<td>85</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00B4</td>
<td>86</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00B8</td>
<td>87</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00C4</td>
<td>88</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00CD</td>
<td>89</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00D0</td>
<td>90</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00D6</td>
<td>91</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00DE</td>
<td>92</td>
<td>cmp dx,Xold</td>
</tr>
<tr>
<td>00F0</td>
<td>93</td>
<td>ret ; Exit</td>
</tr>
<tr>
<td>00F1</td>
<td>94</td>
<td>end</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETED. NO ERROR(S) FOUND.
MCS®-96
Analog Acquisition Primer

DAVID P. RYAN
INTEL CORPORATION
THE MCS®-96 ANALOG ACQUISITION PRIMER

INTRODUCTION

As technology advances, embedded control applications continue to reduce chip-count and demand microcontrollers with increased features to assist system-cost reduction. Since every embedded control application interfaces with the physical world, and the physical world is an analog process, it was inevitable that microcontrollers would include integrated analog acquisition capabilities.

The first such integration of standard microcontroller and A/D converter occurred on Intel's 8022 in 1978. This opened the door to cost reduction of high volume applications that required analog inputs. The device fit well into applications that needed processing of analog data. But this chip, with its 8-bit CPU, could not perform in high-end applications requiring analog inputs, or in applications that had computationally demanding analog tasks.

With the introduction of the MCS®-96 family of 16-bit microcontrollers in 1982, the combined CPU and A/D performance became available to greatly reduce the system cost of mid- and high-performance embedded control applications. These are applications which were customarily implemented with 16-bit microprocessor chip-sets teamed with analog acquisition chip sets.

There are less obvious avenues for system cost reduction when a 16-bit CPU is teamed with an on-chip analog acquisition system. For example, closed-loop servo control had been implemented almost exclusively by using analog methods. When an MCS-96 device is designed into such an application, it is not only replacing a microcontroller or microprocessor, but it also replaces closed-loop analog circuitry which never before came in contact with the digital system.

To take full advantage of this new level of integration, digital designers must become familiar with analog acquisition, and analog designers must become familiar with digital methods of processing analog signals. This Application Note assists with the first task—understanding of an analog acquisition system.

Designers experienced with analog design, or analog acquisition systems, may find no revelations herein. To those unfamiliar with analog acquisition systems, this Ap Note provides a tutorial on the subject and will serve as a handy reference.

Answering the limitless number of analog circuit design questions is beyond the scope of this Ap Note. Suffice it to say that the effort placed on the design of analog circuits should increase with a decreasing error budget.

At a minimum, the applications literature of op-amp manufacturers and analog design manuals are a good place to start. Furthermore, the applications literature of monolithic analog acquisition system manufacturers should be consulted since the suggestions presented therein are largely transportable to any A/D system.

This Ap Note is organized in the following sections. The components of an analog acquisition system and the errors associated with each is first explained. Then, interfacing suggestions and ideas for getting more resolution are presented. Finally, a set of appendices provides back-up information, a bibliography, actual converter data and some program listings.

The definitions of terms used, and the examples presented, are drawn from the body of applications literature publicly available on the components of an analog acquisition system. There is usually no single meaning for a particular term or specification used to describe analog acquisition. However, there is, in most cases, a generally accepted definition which is most often used. To the extent possible, we have adopted the most used definition. To avoid any ambiguity, Appendix A lists the dictionary of terms as used to refer to the analog acquisition systems of MCS-96 devices.

For any users of an MCS-96 analog acquisition system (experienced or not), this document contains very useful information. It should be considered mandatory reading in addition to the latest Embedded Controller Handbook and MCS-96 data sheet for the actual device in use prior to the actual design.
WHAT IS AN ANALOG ACQUISITION SYSTEM?

An analog acquisition system is a collection of individual units which, when logically configured, form a system capable of converting an analog input to a digital value.

The typical components of an Analog Acquisition Unit (Figure 1) include an Analog-to-Digital Converter (A/D), a Sample-and-Hold (S/H) and an Analog Multiplexer (MUX). The A/D converts the infinitely varying analog voltage present on the S/H into a digital representation for use by the digital system. The S/H is required so a "snapshot" of a changing analog input can be stored for conversion by the A/D. The MUX is used to leverage the investment in the A/D by allowing a large number of isolated analog input channels to use the same converter.

The conversion result of an MCS-96 device is a 10-bit ratiometric representation of the input voltage. This produces a stair-stepped transfer function when the output code is plotted versus input voltage. See Figure 2.

The resulting digital codes can be taken as simple ratiometric information, or they can be used to provide information about absolute voltages or relative voltage changes on the inputs. The more demanding the application is on the A/D converter, the more important it is to fully understand the converter's operation. For simple applications, knowing the absolute error of the converter is sufficient. However, controlling a closed loop with analog inputs necessitates a detailed understanding of an A/D converter's operation and errors.

The errors inherent in an analog-to-digital conversion process are many: quantizing error; zero offset; full-scale error; differential non-linearity; and non-linearity. These are "transfer function" errors related to the A/D converter. In addition, the S/H and MUX may induce channel dissimilarities and sampling error (described later).

Fortunately, one "Absolute Error" specification is available which describes the sum total of all deviations between the actual conversion process and an ideal converter. The various sub-components of error are, however, important in many applications. These error components are described in Appendix A and in the text below where ideal and actual converters are compared.

A/D Converter

There are at least three well-recognized methods for converting an analog voltage to a digital value—flash, dual slope and successive approximation.

Flash A/Ds are the fastest, and most expensive converters for a given accuracy. Flash converters typically resolve bits of the result in parallel to achieve fast conversions. Flash converter speeds are measured in tens-of-nanoseconds.

Dual slope converters are the slowest, but most accurate. Dual slope conversion is rather insensitive to noise on the input, but conversion times are measured in milliseconds.

Successive approximation converters provide a balanced tradeoff between speed and accuracy. Successive approximation conversion times are measured in tens-of-microseconds, and converter implementations are very economical for a given accuracy.
MCS-96 converters use successive approximation. A successive approximation conversion is performed by comparing a sequence of reference voltages to the analog input in a binary search for the reference voltage that most closely matches the input. The $\frac{1}{2}$ full-scale reference voltages is the tested first. This corresponds to a 10-bit result where the most significant bit is zero, and all other bits are ones (0111 1111 11b). If the analog input is less than the test voltage, bit 10 of the result is left a zero, and a new test voltage of $\frac{1}{2}$ full-scale (00 11 1111 11b) is tried. If this test voltage is lower than the analog input, bit 9 of the result is set and bit 8 is cleared for the next test (0101 1111 11b). This binary search continues until 10 tests have occurred, at which time the valid 10-bit conversion result resides in a register where it can be read by software.

The voltages used during the binary search are generated from an internal Digital-to-Analog Converter similar to Figure 3. The figure shows eight resistors being used as a three-bit D to A. The first resistor tap is taken from the center of the first resistor to guarantee that a zero input voltage will always output a zero code. Each successive tap then provides a reference voltage $V_{REF}/8$ (one LSB) from the previous tap. When the analog input is above the voltage of the seventh tap, the A/D will resolve to its full-scale value of 111b. Therefore, an eighth tap is not needed, and the A/D's 110b to 111b code transition will occur when $V_{ANIN}$ equals $V_{REF} - 1\frac{1}{2}$ LSB.

The first error seen in this process is unavoidable, and results from the conversion of a continuous voltage to
Figure 3. A Three-Bit D-to-A

An integer digital representation. This error is called quantizing error, and is always ±0.5 LSB. Quantizing error is the only error seen in a perfect A/D converter, and is obviously present in actual converters. Figure 2 shows the transfer function for an ideal 3-bit A/D converter (i.e. the Ideal Characteristic).

Note that in Figure 2 the Ideal Characteristic possesses unique qualities: it’s first code transition occurs when the input voltage is 0.5 LSB; it’s full-scale code transition occurs when the input voltage equals the full-scale reference minus 1.5 LSB; and it’s code widths are all exactly one LSB. These qualities result in a digitization without offset, full-scale or linearity errors. In other words, a perfect conversion.

Figure 4 shows an Actual Characteristic of a hypothetical 3-bit converter which is not perfect. When the Ideal Characteristic is overlaid with the imperfect characteristic, the actual converter is seen to exhibit errors in the location of the first and final code transitions and code widths. The deviation of the first code transition from ideal is called “zero offset”. The deviation of the final code transition from ideal is “full-scale error”.

The deviation of the code widths from ideal causes two types of errors. Differential Non-Linearity and Non-Linearity. Differential Non- Linearity is a local linearity error measure, whereas Non-Linearity is an overall linearity error measure. For example, Figure 5a shows a transfer function with a large differential non-linearity and a little non-linearity. In contrast, Figure 5b shows a characteristic with small differential errors but a large overall linearity error.

Differential Non-Linearity is the degree to which actual code widths differ from the ideal width. Differential Non-Linearity gives the user a measure of how much the input voltage may have changed in order to produce a one count change in the conversion result.

If the absolute value of an input voltage is less important than the amount that the input changes, the differential non-linearity (DNL) specification of a converter is very important. For example, if the differential non-linearity of a converter is less than ±0.5 LSB, a one count change in the digital result means that the input voltage changed at most 1.5 LSB (1 LSB ideal ± 0.5 LSB DNL). This is a much more accurate description of the input voltage change than would be available if the differential non-linearity of the converter was not known.
Figure 4. Actual and Ideal Characteristics

Figure 5. Types of Linearity Errors
Non-Linearity is the worst case deviation of code transitions from the corresponding code transitions of the Ideal Characteristic. Non-Linearity describes how much Differential Non-Linearity could add to produce an overall maximum departure from a linear characteristic.

If the Differential Non-Linearity errors are large enough, it is possible for an A/D converter to miss codes or exhibit non-monotonicity. Neither behavior is desirable in a closed-loop system. A converter has no missed codes if there exists for each output code a unique input voltage range that produces that code only. A converter is monotonic if every subsequent code change represents an input voltage change in the same direction. Figure 6a shows a converter with missed codes. Figure 6b shows a non-monotonic converter.

Differential Non-Linearity and Non-Linearity are quantified by measuring the Terminal Based Linearity Errors. A Terminal Based Characteristic results when an Actual Characteristic is shifted and scaled to eliminate zero offset and full-scale error (see Figure 7). The Terminal Based Characteristic is similar to the Actual Characteristic that would be seen if zero offset and full-scale error were externally trimmed away. In practice, this is done by using input circuits which include gain and offset trimming. (See the Application Hints section for more details.)

An often overlooked characteristic of A/D converters is that code transitions do not really occur instantaneously at some finite set of input voltages. Specific code transitions can be analyzed by doing repeated conversions around the transition point using a high accuracy input voltage. When this is done, we find that there is actually a range of voltages around code transitions where both the lower and upper codes occur for repeated conversions on the same input voltage.

Figure 8 shows this “repeatability” error. At the lower end of the region of repeatability error the lower code is most prevalent, but the upper code will occur in a small percentage of the conversion attempts. As the input voltage increases slightly, a point is reached where both lower and upper codes occur with 50 percent probability. As the input voltage moves slightly higher, the upper code occurs most often with the lower code showing up in a small percentage of conversions.

The repeatability error is due to the fundamental ability of the comparator in the A/D to resolve very similar voltages. Random noise also contributes to repeatability errors. On MCS-96 devices, the width of the region of repeatability error has been found to be typically 1 mV to 1.25 mV. Since this error is specified, all other errors are specified assuming the code transitions occur at the voltage where adjacent codes are equally likely.

The Multiplexer

The eight channel multiplexer is implemented as a collection of eight MOS switches. Only one of eight can be closed at any instant in time. Figure 1 shows the multiplexer with the switches acting as resistors when closed and as small parasitic capacitors when open. The input protection devices on the analog input pins are also considered a part of the multiplexer.
The resistance of a closed switch is typically 1K to 2K ohms and the D.C. leakage due to the input protection is typically 3 microamps maximum. Both values depend upon the process used and day-to-day fabrication variations. The channel resistance and the D.C. leakage can also vary from channel-to-channel on the same device. These variations can be seen in the conversion process and are described by the channel-to-channel matching specification.

Channel-to-channel matching specifies the input voltage differences induced by mismatched elements of the multiplexer. This error is quantified by measuring the difference between the input voltages necessary to cause the same code transition to occur through different multiplexer channels under identical test conditions.

Matching errors are more complex than a simple voltage offset between channels, and thus are difficult to externally cancel. Fortunately, multiplexer channels typically match to within one millivolt.

A multiplexer that has the potential to short two inputs together is not very attractive. To keep this from happening, the circuitry that selects the active channel is designed to guarantee that all channels are deselected before a new channel is selected. Thus, the multiplexer is said to be Break-Before-Make.

In addition to Break-Before-Make channel selection, an analog multiplexer must be able to keep deselected channels isolated from the selected channel. As shown in Figure 1, there are parasitic capacitances coupling every deselected channel to the multiplexer output. The quantification of coupling is called Off-Isolation. Off-isolation is the multiplexer's ability to attenuate signals on deselected channels.
Sample-and-Hold

The sample-and-hold of an analog acquisition system can be built using an analog switch and a sample capacitor. As with the multiplexer, there is also a parasitic capacitance coupling the switch input to the sample capacitor when the switch is open (Figure 1).

The resistance of the sample-and-hold switch combines with the series resistance of the multiplexer to impede the current necessary to charge the sample capacitor. For example, with a 5K ohm total input resistance from the pin to the 2 pf sample capacitor, the RC time constant is 10 nS (2 pf × 5K ohms).

During the one microsecond that the sample capacitor is connected to the input, 100 time constants elapse (1 microsecond/10 nS). This means that the sample capacitor is 100 percent of the voltage on the input pin (1 - e^{-100}), assuming a zero source impedance.

If a source impedance of 2K ohms is assumed, the RC time constant of the sampling process would be 14nS (7K ohms × 2 pf). Thus, 71.4 time constants would pass in one microsecond resulting in the sample capacitor being charged to within 99.9 percent of its final value. Source impedances above 2K ohms would begin to degrade the conversion accuracy due to D.C. leakage (described later).

Figure 9 shows the actual input voltage and the sampled voltage approaching the input voltage. Once the sample-and-hold switch closes, the sample window begins. The sample window extends for four state times and ends with the sample-and-hold switch opening on MCS-96 devices (except 8X9X-90, which is 8 state times and has no sample-hold). Figure 9 also shows the sample delay, which is the delay from the time a start conversion signal is generated to the time a conversion process begins.

It is important to understand the uncertainties associated with the timing of the sample-and-hold. Digital signal processing algorithms rely upon the “spectral purity” of the sampling process. If the sample window jumps around with respect to the start conversion signal, or if the start conversion signal cannot be generated at precise times, consecutive samples of input data will not be equally spaced in time (i.e. sampling will be spectrally impure).

The MCS®-96 Conversion Sequence

The MCS-96 Analog Acquisition System includes an eight channel analog multiplexer, sample-and-hold circuit and 10-bit analog to digital converter (Figure 10). An MCS-96 device can therefore select one of eight analog inputs, sample-and-hold the input voltage and convert the voltage into a digital value. Each conversion takes 22 microseconds (8097BH), including the time required for the sample-hold (with XTAL1 = 12 MHz). The method of conversion is successive approximation.

To improve the spectral purity of the sampling in digital signal processing applications, sequential MCS-96 start conversion signals can be generated with less than 50 nanoseconds of jitter using the HSO unit. The sample delay and sample time are also a constant number of state times to within 50 nanoseconds each.

Once the sample window closes, it is desired that all further changes on any input channel be isolated from the sample capacitor. The multiplexer’s off-isolation is responsible for isolating deselected channels, while the sample-and-hold switch must attenuate changes on the selected channel. This source of error is described as Feedthrough. Feedthrough is quantified as the ability of the sample-and-hold to reject unwanted signals on its input.

Other factors that affect a real A/D Converter system include sensitivity to temperature. Temperature sensitivities are described by the change in typical specifications with a change in temperature.
The conversion process is initiated by the execution of HSO command OFH, or by writing a one to the GO Bit in the A/D Control Register. Either activity causes a start conversion signal to be sent to A/D control logic. If an HSO command was used, the conversion process will begin when Timer 1 increments. This aids applications attempting to approach spectrally pure sampling, since successive samples spaced by equal Timer 1 delays will occur with a variance of about ±50 ns (assuming a stable clock on XTAL1). However, conversion initiated by writing a one to the ADCON register GO Bit will start within three state times after the instruction has completed execution, resulting in a variance of about 0.75 μs (XTAL1 = 12 MHz).

Once the A/D unit receives a start conversion signal, there is a one state time delay before sampling (sample delay) while the successive approximation register is reset and the proper multiplexer channel is selected. After the sample delay, the multiplexer output is connected to the sample capacitor and remains connected for four state times (sample time). After this four state time “sample window” closes, the input to the sample capacitor is disconnected from the multiplexer so that changes on the input pin will not alter the stored charge while the conversion is in progress. The sample delay and sample time uncertainties are each approximately ±50 ns, independent of clock speed.

To perform the actual analog-to-digital conversion the MCS-96 implements a successive approximation algorithm. The converter hardware consists of a 256-resistor ladder, a comparator, coupling capacitors and a 10-bit successive approximation register (SAR) with logic that guides the process. The resistor ladder provides 20 mV steps (VREF = 5.12V), while capacitive coupling is used to create 5 mV steps within the 20 mV ladder voltages. Therefore, 1024 internal reference voltages are available for comparison against the analog input to generate a 10-bit conversion result. Appendix B contains a detailed description of the method used to generate 1024 voltages from a 256-resistor chain.

The total number of state times required for a 10-bit conversion varies from one MCS-96 version to the next. Attempting to short-cycle the 10-bit conversion process by reading A/D results before the done bit is set may work on some versions of MCS-96 devices, however it is not recommended. Short-cycling is not tested, nor is it guaranteed. Furthermore, it may not work on future MCS-96 devices.
APPLICATION HINTS

The analog signals that must be converted by an analog acquisition system vary widely. The analog input may arrive at the controller as a voltage or current. The range may be 0 to 1 volt or ±30 volts, or some other arbitrary range. The input may be linear, logarithmic, non-linear, or perturbated in some bizarre fashion. Although interfacing to such signals could be considered an art form, some simple suggestions are contained in this section.

Analog Inputs

The external interface circuitry to an analog input is highly dependent upon the application, and can impact converter characteristics. In the external circuit’s design, important factors such as input pin leakage, sample capacitor size and multiplexer series resistance from the input pin to the sample capacitor must be considered.

![Figure 11. Idealized A/D Sampling Circuitry](image)

![Figure 12. Suggested A/Q Input Circuit](image)

For the 8096BH, these factors are idealized in Figure 11. The external input circuit must be able to charge a sample capacitor (Cs) through a series of resistance (Rl) to an accurate voltage given a D.C. leakage (IL). On the 8096BH, Cs is around 2 pF, Rl is around 5 KΩ and IL is specified at 3 μA maximum. In determining the source impedance Rs, VBIAS is not important.

External circuits with source impedances of 1 KΩ or less will be able to maintain an input voltage within a tolerance of about ±0.61 LSB (1.0 KΩ × 3.0 μA = 3.0 mV) given the D.C. leakage. Source impedances above 2 KΩ can result in an external error of at least one LSB due to the voltage drop caused by the 3 μA leakage. In addition, source impedances above 25 KΩ may degrade converter accuracy as a result of the internal sample capacitor not being fully charged during the 1 μs (12 MHz clock) sample window.

Placing an external capacitor on each analog input will reduce the sensitivity to noise, as the capacitor combines with source resistance in the external circuit to form a low-pass filter. In practice, one should include a small series resistance prior to an external low leakage capacitor on the analog input pin and choose the largest capacitor value practical, given the frequency of the signal being converted. This provides a low-pass filter on the input, while the resistor will also limit input current during over-voltage conditions.

Figure 12 shows a simple analog interface circuit based upon the discussion above. The circuit in the figure also provides limited protection against over-voltage conditions on the analog input (limits to 2.6 mA with 270Ω (0.7/270)). The circuit induces leakage from the diodes, which should be kept small.

The wide range of possible analog environments that must be interfaced to, or the existence of stringent accuracy requirements, makes the consideration of alternative input buffer configurations necessary. The most popular input buffer is a single op-amp in the non-inverting or inverting configurations of Figure 13.

In the non-inverting circuit of Figure 13 (a), the analog input is scaled by the buffer gain to output 5 volts when the input is at its maximum positive input. When the buffer input is 0 volts, the output will also be 0 volts.

In the inverting circuit of Figure 13 (b), a reference equal to the maximum possible input voltage is placed on the non-inverting input of the op-amp and the actual analog input is placed on the inverting input. The output voltage of the buffer is then proportional to the deviation of analog input from its maximum possible value. For example, when the analog input equals VMAX, the buffer output will equal 0 zero volts. When the analog input equals its minimum value, the buffer output equals 5 volts. The digital result from the A/D converter might, of course, have to be complemented before being used.

The circuits of Figure 13 show only feedback resistors that set the gain of the buffer. In practice, it will often be necessary to include offset adjustments, gain trimming, temperature or frequency stability compensation, or components to build an active filter.

Figure 14 depicts a generalized non-inverting input buffer that offsets the analog input and scales the input.
to a 5 volt range. The course offset is set by the ratio of $R_{BIG1}$ and $R_{BIG2}$, while offset fine tuning is done by adjusting $R_{TRIM}$. The course gain is set by the ratio of $R_G1$ and $R_G2$ while gain trimming is done with $R_{GTRIM}$.

The procedure for nulling offset and gain is simple. First, a voltage is applied to $V_{IN}$ which corresponds to the ideal first code transition of the A/D. $R_{TRIM}$ is adjusted so that 50 percent of the conversion results are 0 while 50 percent are 1. Second, a voltage is applied to $V_{IN}$ which corresponds to the ideal final code transition of the A/D converter. $R_{GTRIM}$ is then adjusted until 50 percent of the conversion results are 3FEH and 50 percent are 3FFH. Once this adjustment is complete, the converter zero offset and full-scale errors are nulled, and could be ignored (except for temperature variation). This allows the system to rely upon the tighter, more descriptive converter specifications for Terminal Based Non-Linearity and Differential Non-Linearity.

Analog References

Reference supply levels strongly influence the absolute accuracy of the conversion. For this reason, it is recommended that the ANGND pin be tied to a clean ground, as close to the power supply as possible. Bypass capacitors should also be used between $V_{REF}$ and ANGND. ANGND should be connected to $V_{SS}$ only at the chip. $V_{REF}$ should be well regulated and used only for the A/D converter. The $V_{REF}$ supply can be between 4.5V and 5.5V and needs to be able to source around 5 mA. Figure 15 shows all of these connections.

Note that if only ratiometric information is desired, $V_{REF}$ can be connected to $V_{CC}$. In addition, if the A/D converter is not being used, $V_{REF}$ must be connected to $V_{CC}$ and ANGND to $V_{SS}$ for Port0 to work as a digital port.
Getting More Resolution

Given that the A/D converter can convert an analog input ranging from 0 volts to 5 volts into 1024 steps of 5 millivolts each, the desire for more resolution can come from three basic needs – need extra LSB, need extra MSB, need BOTH.

The configuration shown in Figure 16 can be used to solve each of the “more resolution” problems. This setup requires the use of two input channels with different offsets and gains.

When the 5 millivolt step size of the A/D is too large for the application requirements, but the 5 volt range is sufficient, the system needs an “extra LSB”. For example, an application requiring 2.5 millivolt steps over a 5 volt range needs an 11-bit conversion result. The 11th bit needs to be added to the least significant side of the 10-bit result (the “right”). This can be achieved using the circuit of Figure 16.

If both channels are set for a gain of 2, with channel 1 offset to 2.5 volts, the 5 volt input range is split into 2.5 volt ranges that are amplified by two before being input to the A/D. While \( V_{IN} \) is between 0 and 2.5 volts, channel 0 will be providing a proportional voltage between 0 volts and 5 volts to the A/D converter. Channel 1 will be clamped at 5 volts. When \( V_{IN} \) rises above 2.5 volts, channel 1 will begin to output a proportional voltage between 0 volts and 5 volts to the A/D converter and channel 0 will be clamped at 5 volts. Using this method, an 11-bit (2048 step) result is created with 2.5 millivolt steps (i.e. an extra LSB).

It is useful to note that only one conversion per sample will be required if the software keeps track of which channel is active. The only time that two conversions will be required for one sample is when the voltage crosses the midpoint.

The second reason that “more resolution” is requested is the need for an “extra MSB”. When the converter’s input voltage range is too small (5 volts when 10 volts is needed), but 5 millivolt steps over the actual input voltage range is sufficient, an extra bit is needed on the most significant (“left”) side of the 10-bit result. The circuit of Figure 16 can also be used, with different gains and offsets, to satisfy this extra MSB need by splitting the 10 volt range into 5 volt ranges.

If both channels of Figure 16 are set for unity gain, and channel 1 is offset to 5 volts, an 11-bit conversion result with 5 millivolt steps is available. While \( V_{IN} \) is in the lower half of its range (0 volts to 5 volts), channel 0 will be active. While \( V_{IN} \) is in the upper half of its range (5 volts to 10 volts), channel 1 will be active. Thus, an extra MSB is created.

For applications requiring multiple extra bits of result, the solutions can become more “elegant” (i.e. elaborate). However, it is profitable to first squeeze the most out of the now familiar circuit in Figure 16.

Assume that the analog input, \( V_{IN} \), ranges from 0 volts to 10 volts, and it is desired to measure this range in 2.5 millivolt steps. This requires two extra bits of result – one extra MSB and one extra LSB. A simple extrapolation of the preceding discussion of creating extra bits might have the designer planning to tie up four channels of the multiplexer needlessly. Needlessly, that is, if the application is a typical control application where the high accuracy requirements are only important in the “normal” operating range of the process. Outside of the normal operating range is the “possible” operating range which must be measured, but with less stringent requirements.

Since the requirements of the normal range set the necessary LSB weight, and the extent of the possible range sets the maximum voltage span, it follows that only two channels need to be used (Figure 16). Channel 0 would be set with a gain that compressed the possible \( V_{IN} \) range to 5 volts, while channel 1 would be offset to the normal operating range and would have a gain of two to expand this region of critical interest. With this ap-
approach, 100 percent of the normal operating range is digitized in 2.5 millivolt steps, while 100 percent of the possible range is digitized in 10 millivolt steps.

Unfortunately, not all high resolution applications can be described as a process with a small region of in-control operation, where the process is out-of-control outside of that small region. For example, it is necessary to measure airflow in an engine controlling carburetion. The airflow at idle is likely to be several orders-of-magnitude lower than the airflow at full RPM. The process needs to be in tight control over the entire range, not only when the engine is at half-speed.

When it is desired to measure a process with a fixed percent of error throughout a range spanning several orders-of-magnitude, a non-linear input buffer becomes attractive. For example, assume that the analog signal that needs to be digitized can vary from 1 millivolt to 25 volts and describes a physical process that must be represented digitally with 1 percent error at any point in the possible input range. A linear solution to this application would require a converter with a 10 microvolt LSB (1% × 1 mV), and a resolution of 22 bits (25 V/10 microvolts). This is clearly undesirable.

The use of a log input buffer to compress the 25 volt range logarithmically to 5 volts would satisfy the application requirements. The input would range from 1 millivolt to 25 volts with the output ranging from 0 volts to 5 volts proportionally to the log of \( V_{IN}/1mV \). Each one-percent change in the input voltage would change the output voltage by 5 millivolts (one count). The antilog could be taken in software using a lookup table, or the control calculations could be performed in a log base.

Simple inexpensive log-amps can be built as in Figure 17, or high-accuracy, self-contained log-amps can be purchased. Which is chosen depends upon the application tradeoffs of price and performance.

Other techniques become available for consideration in systems that have slow sample rate requirements, but very high resolution requirements. In addition to the methods described above, which require external hardware, software filtering or other post-processing of the conversion results can be productive. Each method relies upon the ability to sample the analog input much faster than the system requires an analog input.

When resolution is limited by filterable noise, perhaps the most straightforward approach to post-processing is to oversample the input by a factor of \( N \) and digitally low-pass filter the data (i.e. weighted rolling average). A result would be reported to the rest of the system every \( N \) samples (Figure 18). A low-pass filter can increase the signal-to-noise ratio (SNR) by a factor of \( N \) (see bibliography). However, care must be taken to be certain that the input voltage varies slowly with respect to the sampling rate.
Another approach to creating more resolution is called "synchronized dither". Figure 19 shows an input voltage that is constant somewhere between two code transition points. This input is "dithered" by adding a small periodic waveform ($\frac{1}{4}$ LSB steps) to the input while performing an A/D conversion synchronized with each dither step. Every time the dither completes a full cycle, the eight conversion results are averaged to form one digitized value. Since the dither is periodic and symmetrical about 0 volts, its average impact on the input voltage is 0 volts.

The creation of extra resolution can be seen with the example shown in Figure 19. Without dither, the input voltage would always convert to 37H. With dither, one-eighth of the conversions would be 38H and $\frac{7}{8}$ of the conversions would be 37H. If every eight conversions were averaged, the result would be $37H + \frac{1}{8}$ LSB. The possible results given a four level dither, where the input voltage was always within the 37H code width, would be

$$
\begin{align*}
36H + \frac{1}{8} \\
36H + \frac{7}{8} \\
37H + 0 \\
37H + \frac{1}{8} \\
37H + \frac{7}{8}
\end{align*}
$$

Hence, four new levels exist (two bits).

Dither will only create more resolution up to the limit of the A/D converter comparator's ability to distinguish voltages. Since MCS-96 converter repeatability error is typically around 1 millivolt to 1.25 millivolts, $\frac{1}{4}$ LSB dither is the practical limit if no other processing is done. Figure 20 shows a simple method by which the input voltage could be dithered under software control.

While only a few of the more obvious interfacing techniques were described here, there are as many innovative interfacing tricks as there are designers.

**CONCLUSION**

This application note provides a fundamental understanding of MCS-96 analog acquisition for the digital designer. Since answering the limitless number of analog circuit design questions is beyond the scope of this document, it is expected that analog design manuals and the large body of publicly available applications literature will be consulted for detailed design hints. Furthermore, the applications literature of monolithic analog acquisition system manufacturers should be consulted since the suggestions presented therein are largely transportable to any A/D system.
Figures 2, 4 and 7 display many of these terms.

**ABSOLUTE ERROR**—The maximum difference between corresponding actual and ideal code transitions. Absolute Error accounts for all deviations of an actual converter from an ideal converter.

**ACTUAL CHARACTERISTIC**—The characteristic of an actual converter. The characteristic of a given converter may vary over temperature, supply voltage, and frequency conditions. An Actual Characteristic rarely has ideal first and last transition locations or ideal code widths. It may even vary over multiple conversions under the same conditions.

**BREAK-BEFORE-MAKE**—The property of a multiplexer which guarantees that a previously selected channel will be deselected before a new channel is selected. (e.g. the multiplexer will not short inputs together.)

**CHANNEL-TO-CHANNEL MATCHING**—The difference between corresponding code transitions of actual characteristics taken from different channels under the same temperature, voltage and frequency conditions.

**CHARACTERISTIC**—A graph of input voltage versus the resultant output code for an A/D converter. It describes the transfer function of the A/D converter.

**CODE**—The digital value output by the converter.

**CODE CENTER**—The voltage corresponding to the midpoint between two adjacent code transitions.

**CODE TRANSITION**—The point at which the converter changes from an output code of Q, to a code of Q + 1. The input voltage corresponding to a code transition is defined to be that voltage which is equally likely to produce either of two adjacent codes.

**CODE WIDTH**—The voltage corresponding to the difference between two adjacent code transitions.

**CROSSTALK**—See “Off-Isolation”.

**D.C. INPUT LEAKAGE**—D.C. Leakage current of an analog input pin.

**DIFFERENTIAL NON-LINEARITY**—The difference between the ideal and actual code widths of the terminal based characteristic of a converter.

**FEEDTHROUGH**—Attenuation of a voltage applied on the selected channel of the A/D converter after the sample window closes.

**FULL-SCALE ERROR**—The difference between the expected and actual input voltage corresponding to the full-scale code transition.

**IDEAL CHARACTERISTIC**—A characteristic with its first code transition at \( V_{IN} = 0.5 \) LSB, its last code transition at \( V_{IN} = (V_{REF} - 1.5 \) LSB) and all code widths equal to one LSB.

**INPUT RESISTANCE**—The effective series resistance from the analog input pin to the sample capacitor.

**LSB - LEAST SIGNIFICANT BIT**—The voltage value corresponding to the full-scale voltage divided by \( 2^n \), where \( n \) is the number of bits of resolution of the converter. For a 10-bit converter with a reference voltage of 5.12 volts, one LSB is 5.0 mV. Note that this is different than digital LSBs, since an uncertainty of two LSBs, when referring to an A/D converter, equals 10 mV. (This has been confused with an uncertainty of two digital bits, which would mean four counts, or 20 mV.)

**MONOTONIC**—The property of successive approximation converters which guarantees that increasing input voltages produce adjacent codes of increasing value, and that decreasing input voltages produce adjacent codes of decreasing value.

**NO MISSED CODES**—For each and every output code, there exists a unique input voltage range which produces that code only.

**NON-LINEARITY**—The maximum deviation of code transitions of the terminal based characteristic from the corresponding code transitions of the actual characteristic of a converter.

**OFF-ISOLATION**—Attenuation of a voltage applied on a deselected channel of the A/D converter. (Also referred to as Crosstalk.)
REPEATABILITY—The difference between corresponding code transitions from different actual characteristics taken from the same converter on the same channel at the same temperature, voltage and frequency conditions.

RESOLUTION—The number of input voltage levels that the converter can unambiguously distinguish between. Also defines the number of useful bits of information which the converter can return.

SAMPLE DELAY—The delay from receiving the start conversion signal to when the sample window opens.

SAMPLE DELAY UNCERTAINTY—The variation in the Sample Delay.

SAMPLE TIME—The time that the sample window is open.

SAMPLE TIME UNCERTAINTY—The variation in the sample time.

SAMPLE WINDOW—Begins when the sample capacitor is attached to a selected channel and ends when the sample capacitor is disconnected from the selected channel.

SUCCESSIVE APPROXIMATION—An A/D conversion method which uses a binary search to arrive at the best digital representation of an analog input.

TEMPERATURE COEFFICIENTS—Change in the stated variable per degree centigrade temperature change. Temperature coefficients are added to the typical values of a specification to see the effect of temperature drift.

TERMINAL BASED CHARACTERISTIC—An Actual Characteristic which has been rotated and translated to remove zero offset and full-scale error.

VCC REJECTION—Attenuation of noise on the VCC line to the A/D converter.

ZERO OFFSET—The difference between the expected and actual input voltage corresponding to the first code transition.
APPENDIX B
CAPACITIVE INTERPOLATION

A successive approximation A/D converter needs an internal D/A converter of the same resolution as the desired A/D result. A 10-bit D/A could have been made using a string of 1024 resistors connected from the analog reference at one end to ground at the other end. Although this would be technically ideal, such a circuit would be enormous. Therefore, a method was developed to generate the needed reference voltages using a small area of silicon so that an on-chip 10-bit A/D converter would be economical.

The method used relies upon a 256-resistor chain to generate reference voltages in 20mV (5.12V/256) steps while two ratioed capacitors are used to capacitively "interpolate" voltages in-between the resistor tap voltages. The area of the 256-resistor chain together with the capacitors is one-fourth the area of the would-be 1024 resistor chain.

Before beginning a detailed description of the capacitive part of the conversion process, it is necessary to understand a few details about the resistor chain.

There are 256 resistors connected in series from the analog reference to analog ground. The actual value of the resistors only impacts the current through the reference pin. If every resistor in the chain is the same value the converter will function properly.

To reduce resistor-to-resistor variation, the chain is folded in half, and then in an accordion fashion to produce a 16 × 16 block of resistors. This minimizes the sensitivity of the array to processing gradients, while also allowing the array to be addressed roughly similar to a 16 × 16 memory array.

As explained earlier, it is desired for the A/D converter to have its first code transition at ½ LSB followed by subsequent code widths 1 LSB wide.

To accomplish this, each resistor is tapped in its center rather than between resistors. For example, the first resistor tap is half-way up the first resistor. This means that the zero resistor tap will output 10mV (20mV/2). When calculating the voltage on a certain resistor tap, you must add 10mV to the product of the tap number and 20mV.

The internal connections while an analog input is being sampled are shown in Figure B1a. Once sampling is complete, the analog input is disconnected and the comparator inputs are no longer clamped to VBIAS (Figure B1b).

During the sample window (Figure B1a), V\text{ANIN} and V\text{OFS} control the amount of charge stored in CA and CB (V\text{OFS} controls the converter offset). Once the sample window closes (Figure B1b), voltages applied to V\text{IN} and V\text{IN2} will add or subtract charge proportional to (V\text{ANIN} - V\text{IN}) on CA and (V\text{OFS} - V\text{IN2}) on CB. Unless a voltage is applied to V\text{IN} and V\text{IN2}. The inverting comparator input of Figure B1b will remain at VBIAS due to the charges on CA and CB. The non-inverting comparator input will always remain at VBIAS and serves as a reference.

If a V\text{IN}, V\text{IN2} combination is applied which causes the non-inverting input to drop below VBIAS the comparator will output a 1 to indicate that the applied voltage was lower than the original V\text{ANIN}. To better understand how the circuit works, Figure B2 shows the superposition analysis used to form the equation for V\text{OUT}, given initial charge on CA and CB and new input voltages V\text{IN} and V\text{IN2}.
Adding the independent effects shown in Figure B2 we have:

\[ V_{OUT} = V_1 + V_2 + V_3 + V_4 \]

\[
V_{OUT} = V_{IN} \left( \frac{C_A}{C_A + C_B} \right) + V_{IN2} \left( \frac{C_B}{C_A + C_B} \right) + V_{AI} \left( \frac{C_A}{C_A + C_B} \right) + V_{BI} \left( \frac{C_B}{C_A + C_B} \right)
\]

\[
V_{OUT} = (V_{IN} + V_{AI}) \frac{C_A}{C_A + C_B} + (V_{IN2} + V_{BI}) \frac{C_B}{C_A + C_B}
\]

(I)

The initial conditions on \( C_A \) and \( C_B \) are set up as shown in Figure B3.

We can see that:

\[ V_{AI} = V_{BIAS} - V_{ANIN} \]

(II)

\[ V_{BI} = V_{BIAS} - V_{OPS} \]

(III)

**Figure B1**

6-215
Substituting II and III into I we get:

\[ V_{\text{OUT}} = (V_{\text{IN}} + V_{\text{BIAS}} - V_{\text{ANIN}}) \frac{C_A}{C_A + C_B} + \]

\[ (V_{\text{IN2}} + V_{\text{BIAS}} - V_{\text{OFS}}) \frac{C_B}{C_A + C_B} \]  \hspace{1cm} (IV)

\( V_{\text{OUT}} \) becomes the input voltage to the comparator which ideally presents no load. The only way to make \( V_{\text{OUT}} \) approach the value of \( V_{\text{BIAS}} \) (after \( V_{\text{BIAS}} \) is removed) is to apply a voltage combination which makes equation IV evaluate to \( V_{\text{BIAS}} \). If we had an infinitely variable internal voltage reference to use, we could just set the reference on \( V_{\text{IN}} \) to the value of \( V_{\text{ANIN}} \) and make \( V_{\text{INZ}} = V_{\text{OFS}} \).

We would then have, from IV:

\[ V_{\text{IN}} = V_{\text{ANIN}}, \quad V_{\text{IN2}} = V_{\text{OFS}} \]

However, using a 256-resistor chain to provide references, we can find a \( V_{\text{IN}}, V_{\text{INZ}} \) combination which can bring \( V_{\text{OUT}} \) close to the value of \( V_{\text{BIAS}} \). The 256-resistor chain provides a reference voltage in 20 mV steps. We can then take separate taps of the resistor chain and connect them to \( V_{\text{IN}} \) and \( V_{\text{IN2}} \). The voltage attached to \( V_{\text{IN}} \) will couple to \( V_{\text{OUT}} \) by a factor of \( \frac{C_A}{C_A + C_B} \) from EQN IV. The voltage attached to \( V_{\text{IN2}} \) will couple to \( V_{\text{OUT}} \) by a factor of \( \frac{C_B}{C_A + C_B} \). The ratio of the impacts on \( V_{\text{OUT}} \) of \( V_{\text{IN}} \) versus \( V_{\text{IN2}} \) is:

\[ \left( \frac{\partial V_{\text{OUT}}}{\partial V_{\text{IN}}} \right) + \left( \frac{\partial V_{\text{OUT}}}{\partial V_{\text{IN2}}} \right) = \frac{8}{9} \]  \hspace{1cm} (8/9)/(1/9) = 8

Therefore, a voltage change on \( V_{\text{IN}} \) will affect the voltage seen at \( V_{\text{OUT}} \) eight times more than the same change placed on \( V_{\text{IN2}} \).
For example, assume the actual input voltage $V_{ANIN}$ was 2.50mV during the sample window. Using Eqn IV, and assuming $V_{BIAS} = 3V$ and $V_{OFS} = 70mV$, we substitute and find:

$$V_{OUT} = (V_{IN} + 2.9975) \times (8/9) + (V_{IN2} + 2.93) \times (1/9)$$

(V)

Using successive approximation, the first trial input voltage attempted corresponds to the digital code 0111 1111 1b (127 $\times$ 20mV + 10mV). This means that the voltage applied to $V_{IN}$ will be the 0111 1111b tap and the voltage applied to $V_{IN2}$ will be the 0110b tap (6 $\times$ 20mV + 10mV = 3 LSB). Substituting these values into Eqn V we have:

$$V_{OUT} = (2.550 + 2.9975) \times (8/9) + (0.130 + 2.93) \times (1/9)$$

$$V_{OUT} = 4.931 + 0.34 = 5.271$$

(V)

Since the 3V reference is lower than $V_{OUT}$ with these inputs, the comparator will output a 0 which is placed in the MSB of the successive approximation register. The next most significant bit of the SAR is then zero'd and the new ladder tap applied to $V_{IN}$. The result of this second comparison, and the subsequent comparisons are shown in Table B1. The C program used to generate Table B1 is listed in Listing B1.

The value selected for $V_{OFS}$ during the sample window may not be obvious. The purpose of $V_{OFS}$ is to inject a constant offset in the sampling process so that the converter's first code transition will occur at 2.5mV.

Using Eqn IV we can quickly see why $V_{OFS}$ is chosen to be the fourth resistor tap ($4 \times 20mV + 10mV = 70mV$). For $V_{ANIN} = 2.5mV$, we want $V_{OUT}$ to evaluate to $V_{BIAS}$ when the SAR is OH.

$$V_{OUT} = \frac{(0.20 mV + 10 mV) + (V_{BIAS} - 2.5 mV)}{(8/9)} + \frac{(0.20 mV + 10mV) + (V_{BIAS} - 70 mV)}{(1/9)}$$

$$V_{OUT} - V_{BIAS} = 7.5 mV \times (8/9) - 60 mV \times (1/9) = 0$$

Therefore, if $V_{OFS} = 70 mV$, the converter’s first code transition will be when $V_{ANIN} = 2.5 mV$.

### Table B1. Conversion Simulation

| A to D simulator. (center taps) |  |
|---------------------------------|  |
| $V_{IN} = 0.002500$            |  |
| $V_{CENT} = 3.000000$          |  |
| $V_{OFF} = 0.070000$           |  |
| SAR = 1FFH (511)               | $V_{OUT} = 5.271111$ |
| SAR = FFH (255)                | $V_{OUT} = 4.133333$ |
| SAR = 7FH (127)                | $V_{OUT} = 3.564444$ |
| SAR = 3FH (63)                 | $V_{OUT} = 3.280000$ |
| SAR = 1FH (31)                 | $V_{OUT} = 3.137778$ |
| SAR = FH (15)                  | $V_{OUT} = 3.066667$ |
| SAR = 7H (7)                   | $V_{OUT} = 3.031111$ |
| SAR = 3H (3)                   | $V_{OUT} = 3.013333$ |
| SAR = 1H (1)                   | $V_{OUT} = 3.004444$ |
| SAR = 0H (0)                   | $V_{OUT} = 3.000000$ |
| SAR = 1H (1)                   | which means 0.005000 volts |

6-217
```c
#include "CTYPB.H"
#include "STDIO.H"

/* example invocation lines */
a2dsim 0.0025 3.0 0.07 p
   Vin Vbias Vofs    print to screen and lp
a2dsim 0.0075 3.0 0.07
   Vin Vbias Vofs    print to screen only

int main(int argc, char *argv)
int k;
char *argv[];
{
   /* main */
   FILE *fp = fopen();
doubleinitial_conditions, vin, vout, vcent, voff, v09, v19;
unsigned int sar = 0x3FF;
unsigned int mask = 0x200;
unsigned int count = 0;
unsigned int prlnton;
if (strcmp(argv[0], "run") == 0)
count++;
if ((k != (4 + count)) \& \& (k != (5 + count)))
   {
   printf("invocation error!\n");
   return;
   }
   count++;
sscanf(argv[count++], "%lf", &vin);
sscanf(argv[count++], "%lf", &vcent);
sscanf(argv[count++], "%lf", &voff);
if (count == k)
   prlnton = 0;
else prlnton = 1;
printf("A to D simulator.(center taps)...");
if (prlnton)
   {
   if ((fp = fopen("\brn\", "w")) == 0)
      {
      printf("can't open printer\n");
      return;
      }
   }
   if (prlnton)
      fprintf(fp, "A to D simulator...");
   printf(" with \nVin = %f\n\n\nVcent = %f\n\nVoff = %f\n\n\nV09 = %f\n\nV19 = %f

initial_conditions = ((8.0 / 9.0) * (vcent - vin))
   + ((1.0 / 9.0) * (vcent - voff));
   v09 = 8.0 / 9.0;
v19 = 1.0 / 9.0;
```

Listing B1. A/D Converter Simulator
sar *= mask;
printf("SAR = %3xH (%4d)\n", sar, sar);
if (printon)
    printf(fp, "SAR = %3xH (%4d)\n", sar, sar);
for (count = 0; count < 10; count++)
{
    vout = (v09 + ((double) (sar >> 2)) * 0.02 + 0.01)
        + (v19 + ((double) ((sar & 3) << 1)) * 0.02 + 0.01)
    + initial_conditions;
    if (vout < vcent)
        sar += mask;
    sar *= mask;
    printf("Vout = %s\nSAR = %3xH (%4d)\n", vout, sar, sar);
    if (printon)
        printf(fp, "Vout = %s\nSAR = %3xH (%4d)\n", vout, sar, sar);
}
printf(" which means %f Volts\n", (double) sar * 0.005);
if (printon)
    printf(fp, " which means %f Volts\n", (double) sar * 0.005);
return;
} /* main */

Listing B1. A/D Converter Simulator (Continued)
The following C program listing contains the routines used to calculate A/D performance in the Embedded Controller Applications lab. Most of the routines require floating point arrays to operate upon. In the listings, the array \( x[\] \) contains the input voltages corresponding to each code transition of the converter. The array \( dx[\] \) contains the width of the region in which each code transition of the converter could occur. For example, an input voltage of 0.003V may cause code 0 and code 1 to be equally likely outputs. \( x[0] \) would then contain 0.0030000. However, 0-to-1 code transitions might be observed infrequently through a range of input voltages from 0.0025V to 0.0035V. \( dx[0] \) would then contain 0.0010000 to indicate that there is a 1 millivolt window in which either code could occur. \( x[\] \) and \( dx[\] \) are generated by hardware doing repeated conversions using precision voltage standards to provide the input voltages. The array \( dd[\] \) is used throughout as temporary storage.

Generally, typical data is drawn from \( x[\] \) only. When minimum and maximum data is desired, \( x[\] \) and \( dx[\] \) are used to find the range of possible input voltages that could cause each code. For example, typical zero offset is found by simply subtracting 0.5 LSB from the value of \( x[0] \). But, the minimum and maximum zero offset would be calculated as \( x[0] - 0.5 \text{ LSB} \pm dx[0]/2 \).

The listings are provided to show exactly how performance data is calculated. They are not meant to be compiled by the reader. In fact, they are too incomplete to compile correctly, as some support routines and global data structures are not provided.
```c
#include "\DPR\ADTMAC.H"
#include "\DPR\DORBASE.H"
#include "\DPR\DBBASE.H"
define LSB (pow(\avcc/(pow(2.nbits))))
define FCT (int)\pow(2.nbits) - 2)
#undef min
#undef max
#undef abs

double pow(a, b)
int a, b;

{ /* pow */
  double temp;
  int i;
  temp = 1.0;
  for (i = 1; i <= ((int) b); i++, temp = temp * a)
    return (temp);
} /* pow */

double fabs(a)
double a:

{ if (a < 0)
    return (-a);
else return (a);
}

int min(a, b)
double a, b:

{ if (a < b)
    return (1);
else if (a > b)
    return (2);
else return (0);
}

int max(a, b)
double a, b:

{ return (min(b, a));
}

double typeoff(x, dx)
float x[], dx[]:

{ double pow();
  return (x[0] - (0.5 * LSB));
}

double maxoff(x, dx)
float x[], dx[]:

{ double pow():
  return (x[0] + (dx[0] / 2.0) - 0.5 * LSB);
}

double minoff(x, dx)
```
Listing C1. Error Formulas (Continued)

6-222
Listing C1. Error Formulas (Continued)
for (i = worst = start; i <= stop; i++)
{
    t1 = (x[i] - (dx[i] / 2.0) - oadj) * gadj - (((double) i + 0.5) * LSB);
    t2 = (x[i] + (dx[i] / 2.0) - oadj) * gadj - (((double) i + 0.5) * LSB);
    if (fabs(t1) > fabs(t2))
        dd[i] = t1;
    else dd[i] = t2;
    if (fabs(dd[i]) > fabs(dd[worst]))
        worst = i;
}
return (worst);

int xdat(x, dx, dd, start, stop) /* using x only */
float x[], dx[], dd[];
int start, stop;
{
    int i, worst;
    double pow(), fabs();
    double oadj, gadj;
    double typez(x), typeoff();

    oadj = typeoff(x, dx);
    gadj = 1.0 * ((typez(x, dx) - oadj) / x[stop]);

    worst = start;
    if (start == 0)
    {
        dd[0] = 0.0;
        start++;
    }
    for (i = start; i <= stop; i++)
    {
        dd[i] = (x[i] - oadj) * gadj
                - (x[i] - 1) - oadj) * gadj
                - LSB;
        if (fabs(dd[i]) > fabs(dd[worst]))
            worst = i;
    }
return (worst);

int xdatdixz(x, dx, dd, start, stop) /* using x and dx */
float x[], dx[], dd[];
int start, stop;
{
    int i, worst;
    double pow(), fabs();
    double t1, t2;
    double oadj, gadj;
    double typez(x), typeoff();

    oadj = typeoff(x, dx);
    gadj = 1.0 * ((typez(x, dx) - oadj) / x[stop]);

    worst = start;
    if (start == 0)
    {
        dd[0] = dx[0] / 2.0;
        start++;
    }
    for (i = start; i <= stop; i++)
    {
        t1 = (x[i] - (dx[i] / 2.0) - oadj) * gadj
                - (x[i] - 1) - oadj) * gadj
                - LSB;
        t2 = (x[i] + (dx[i] / 2.0) - oadj) * gadj - (((double) i + 0.5) * LSB);
        if (fabs(t1) > fabs(t2))
            dd[i] = t1;
        else dd[i] = t2;
        if (fabs(dd[i]) > fabs(dd[worst]))
            worst = i;
    }
return (worst);
Listing C1. Error Formulas (Continued)
APPENDIX D
SAMPLE CONVERTER DATA

The following pages include printouts describing the performance of an 8097BH. The data shown is for one device and is provided for illustrative purposes only. Users should only rely upon data sheet specifications for the exact device they are designing with.

Table D1 summarizes many performance measures for one converter at 25°C, 12 MHz, \( V_{CC} = 5.00 \) volts and \( V_{REF} = 5.120 \) volts. Following Table D2 are several error plots that describe Absolute Error, Terminal-based Non-Linearity, Differential Non-Linearity and Repeatability for the test device code-by-code. The y-axis in the plots is the error in volts for each code transition, where code transitions make up the x-axis.

<table>
<thead>
<tr>
<th>Test ID = DOH</th>
</tr>
</thead>
<tbody>
<tr>
<td>sN: 4130 (1022H)</td>
</tr>
<tr>
<td>( T = 25.000000 )</td>
</tr>
<tr>
<td>( V_{CC} = 5.000000 ), ( A_{VCC} = 5.120000 )</td>
</tr>
<tr>
<td>Freq = 12.000000</td>
</tr>
<tr>
<td>Chan. = 3</td>
</tr>
<tr>
<td>States = 188 \ Mode = 0H</td>
</tr>
<tr>
<td>X0.15 1/28/87</td>
</tr>
<tr>
<td>Transition Characterization Parameter Listing</td>
</tr>
<tr>
<td>Large Step = 0.001000 ( V )</td>
</tr>
<tr>
<td>Small Step = 0.000100 ( V )</td>
</tr>
<tr>
<td>Endpoints when (1/100) are wrong</td>
</tr>
<tr>
<td>Center is 50 percent</td>
</tr>
<tr>
<td>Typical Offset Error = -0.001923</td>
</tr>
<tr>
<td>Maximum Offset Error = -0.002460</td>
</tr>
<tr>
<td>Maximum Offset Error = -0.001385</td>
</tr>
<tr>
<td>Typical FS Error = -0.000566</td>
</tr>
<tr>
<td>Maximum FS Error = -0.001254</td>
</tr>
<tr>
<td>Minimum FS Error = -0.000120</td>
</tr>
<tr>
<td>Absolute Error (typ) 40 = 0.004157</td>
</tr>
<tr>
<td>Absolute Error (max) 40 = 0.004795</td>
</tr>
<tr>
<td>Absolute Error (min) 325 = 0.001111</td>
</tr>
<tr>
<td>Diff. Non. Lin. Error (max) 40 = 0.003747</td>
</tr>
<tr>
<td>Diff. Non. Lin. Error (min) FF = -0.001071</td>
</tr>
<tr>
<td>Term. Non. Lin. Error (max) 325 = -0.004102</td>
</tr>
<tr>
<td>Term. Non. Lin. Error (min) 40 = 0.002148</td>
</tr>
<tr>
<td>Maximum Reliability Error 3D1 = 0.001875</td>
</tr>
<tr>
<td>Minimum Reliability Error 3A7 = 0.000974</td>
</tr>
</tbody>
</table>

Resolution is 1024 levels.
### Absolute Error, SN = 4130

<table>
<thead>
<tr>
<th>y_min</th>
<th>y_max</th>
<th>+ 0 -</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.0052</td>
<td>0.0052</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0.002460</th>
<th>0.002460</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.002214</td>
<td>0.002214</td>
</tr>
<tr>
<td>2</td>
<td>0.002257</td>
<td>0.002257</td>
</tr>
<tr>
<td>3</td>
<td>0.002171</td>
<td>0.002171</td>
</tr>
<tr>
<td>4</td>
<td>0.002597</td>
<td>0.002597</td>
</tr>
<tr>
<td>5</td>
<td>0.002291</td>
<td>0.002291</td>
</tr>
<tr>
<td>6</td>
<td>0.002334</td>
<td>0.002334</td>
</tr>
<tr>
<td>7</td>
<td>0.002172</td>
<td>0.002172</td>
</tr>
<tr>
<td>8</td>
<td>0.002579</td>
<td>0.002579</td>
</tr>
<tr>
<td>9</td>
<td>0.002136</td>
<td>0.002136</td>
</tr>
<tr>
<td>A</td>
<td>0.002263</td>
<td>0.002263</td>
</tr>
<tr>
<td>B</td>
<td>0.002219</td>
<td>0.002219</td>
</tr>
<tr>
<td>C</td>
<td>0.002652</td>
<td>0.002652</td>
</tr>
<tr>
<td>D</td>
<td>0.002230</td>
<td>0.002230</td>
</tr>
<tr>
<td>E</td>
<td>0.002280</td>
<td>0.002280</td>
</tr>
<tr>
<td>F</td>
<td>0.002062</td>
<td>0.002062</td>
</tr>
<tr>
<td>10</td>
<td>0.002581</td>
<td>0.002581</td>
</tr>
<tr>
<td>11</td>
<td>0.002263</td>
<td>0.002263</td>
</tr>
<tr>
<td>12</td>
<td>0.002449</td>
<td>0.002449</td>
</tr>
<tr>
<td>13</td>
<td>0.002165</td>
<td>0.002165</td>
</tr>
<tr>
<td>14</td>
<td>0.002570</td>
<td>0.002570</td>
</tr>
<tr>
<td>15</td>
<td>0.002129</td>
<td>0.002129</td>
</tr>
<tr>
<td>16</td>
<td>0.002262</td>
<td>0.002262</td>
</tr>
<tr>
<td>17</td>
<td>0.002192</td>
<td>0.002192</td>
</tr>
<tr>
<td>18</td>
<td>0.002533</td>
<td>0.002533</td>
</tr>
<tr>
<td>19</td>
<td>0.002223</td>
<td>0.002223</td>
</tr>
<tr>
<td>20</td>
<td>0.002383</td>
<td>0.002383</td>
</tr>
<tr>
<td>21</td>
<td>0.002300</td>
<td>0.002300</td>
</tr>
<tr>
<td>22</td>
<td>0.002475</td>
<td>0.002475</td>
</tr>
<tr>
<td>23</td>
<td>0.002268</td>
<td>0.002268</td>
</tr>
<tr>
<td>24</td>
<td>0.002410</td>
<td>0.002410</td>
</tr>
<tr>
<td>25</td>
<td>0.001994</td>
<td>0.001994</td>
</tr>
<tr>
<td>26</td>
<td>0.002741</td>
<td>0.002741</td>
</tr>
<tr>
<td>27</td>
<td>0.002392</td>
<td>0.002392</td>
</tr>
<tr>
<td>28</td>
<td>0.002516</td>
<td>0.002516</td>
</tr>
<tr>
<td>29</td>
<td>0.002392</td>
<td>0.002392</td>
</tr>
<tr>
<td>30</td>
<td>0.002713</td>
<td>0.002713</td>
</tr>
<tr>
<td>31</td>
<td>0.002580</td>
<td>0.002580</td>
</tr>
<tr>
<td>32</td>
<td>0.002262</td>
<td>0.002262</td>
</tr>
<tr>
<td>33</td>
<td>0.002299</td>
<td>0.002299</td>
</tr>
<tr>
<td>34</td>
<td>0.002865</td>
<td>0.002865</td>
</tr>
<tr>
<td>35</td>
<td>0.002534</td>
<td>0.002534</td>
</tr>
<tr>
<td>36</td>
<td>0.002605</td>
<td>0.002605</td>
</tr>
<tr>
<td>37</td>
<td>0.002155</td>
<td>0.002155</td>
</tr>
<tr>
<td>38</td>
<td>0.002841</td>
<td>0.002841</td>
</tr>
<tr>
<td>39</td>
<td>0.002787</td>
<td>0.002787</td>
</tr>
<tr>
<td>40</td>
<td>0.002487</td>
<td>0.002487</td>
</tr>
<tr>
<td>41</td>
<td>0.002733</td>
<td>0.002733</td>
</tr>
<tr>
<td>42</td>
<td>0.002246</td>
<td>0.002246</td>
</tr>
<tr>
<td>43</td>
<td>0.002865</td>
<td>0.002865</td>
</tr>
<tr>
<td>44</td>
<td>0.002534</td>
<td>0.002534</td>
</tr>
<tr>
<td>45</td>
<td>0.002605</td>
<td>0.002605</td>
</tr>
<tr>
<td>46</td>
<td>0.002155</td>
<td>0.002155</td>
</tr>
<tr>
<td>47</td>
<td>0.002841</td>
<td>0.002841</td>
</tr>
</tbody>
</table>

Absolute Error, SN = 4130
Absolute Error, SN = 4130 (Continued)
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B9:</td>
<td>0.002225:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4:</td>
<td>0.002263:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B8:</td>
<td>0.002113:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7:</td>
<td>0.002233:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BB:</td>
<td>0.002172:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BE:</td>
<td>0.002369:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BF:</td>
<td>0.002149:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0:</td>
<td>0.002216:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1:</td>
<td>0.001841:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2:</td>
<td>0.002051:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3:</td>
<td>0.001935:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4:</td>
<td>0.001965:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F5:</td>
<td>0.001729:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6:</td>
<td>0.001979:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F7:</td>
<td>0.001899:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F8:</td>
<td>0.001569:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F9:</td>
<td>0.001718:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FA:</td>
<td>0.001935:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FB:</td>
<td>0.001756:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC:</td>
<td>0.001975:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD:</td>
<td>0.001832:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FE:</td>
<td>0.001920:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FF:</td>
<td>0.001041:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100:</td>
<td>0.002291:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101:</td>
<td>0.002008:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102:</td>
<td>0.002296:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>103:</td>
<td>0.001975:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104:</td>
<td>0.001946:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105:</td>
<td>0.001874:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>106:</td>
<td>0.001884:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>107:</td>
<td>0.001817:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108:</td>
<td>0.002135:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109:</td>
<td>0.001921:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10A:</td>
<td>0.002009:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10B:</td>
<td>0.001832:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10C:</td>
<td>0.001903:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10D:</td>
<td>0.001694:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10E:</td>
<td>0.001086:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10F:</td>
<td>0.001537:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110:</td>
<td>0.001681:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111:</td>
<td>0.001436:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112:</td>
<td>0.001730:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113:</td>
<td>0.001631:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114:</td>
<td>0.001656:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115:</td>
<td>0.001374:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116:</td>
<td>0.001550:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117:</td>
<td>0.001550:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>118:</td>
<td>0.001530:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>119:</td>
<td>0.001411:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11A:</td>
<td>0.001390:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11B:</td>
<td>0.001271:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11C:</td>
<td>0.001321:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11D:</td>
<td>0.001074:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11E:</td>
<td>0.001260:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11F:</td>
<td>0.000614:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120:</td>
<td>0.001401:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121:</td>
<td>0.001052:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>122:</td>
<td>0.001193:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123:</td>
<td>0.001106:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124:</td>
<td>0.001253:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Absolute Error, SN = 4130 (Continued)**
Absolute Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Index</th>
<th>Error Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>161</td>
<td>0.001755</td>
</tr>
<tr>
<td>162</td>
<td>0.001722</td>
</tr>
<tr>
<td>163</td>
<td>0.001706</td>
</tr>
<tr>
<td>164</td>
<td>0.001694</td>
</tr>
<tr>
<td>165</td>
<td>0.001681</td>
</tr>
<tr>
<td>166</td>
<td>0.001630</td>
</tr>
<tr>
<td>167</td>
<td>0.001612</td>
</tr>
<tr>
<td>168</td>
<td>0.001597</td>
</tr>
<tr>
<td>169</td>
<td>0.001580</td>
</tr>
<tr>
<td>170</td>
<td>0.001564</td>
</tr>
<tr>
<td>171</td>
<td>0.001550</td>
</tr>
<tr>
<td>172</td>
<td>0.001528</td>
</tr>
<tr>
<td>173</td>
<td>0.001516</td>
</tr>
<tr>
<td>174</td>
<td>0.001506</td>
</tr>
<tr>
<td>175</td>
<td>0.001500</td>
</tr>
<tr>
<td>176</td>
<td>0.001493</td>
</tr>
<tr>
<td>177</td>
<td>0.001487</td>
</tr>
<tr>
<td>178</td>
<td>0.001481</td>
</tr>
<tr>
<td>179</td>
<td>0.001476</td>
</tr>
<tr>
<td>180</td>
<td>0.001471</td>
</tr>
<tr>
<td>181</td>
<td>0.001465</td>
</tr>
<tr>
<td>182</td>
<td>0.001460</td>
</tr>
<tr>
<td>183</td>
<td>0.001456</td>
</tr>
<tr>
<td>184</td>
<td>0.001452</td>
</tr>
<tr>
<td>185</td>
<td>0.001449</td>
</tr>
<tr>
<td>186</td>
<td>0.001436</td>
</tr>
<tr>
<td>187</td>
<td>0.001433</td>
</tr>
<tr>
<td>188</td>
<td>0.001429</td>
</tr>
<tr>
<td>189</td>
<td>0.001426</td>
</tr>
<tr>
<td>190</td>
<td>0.001411</td>
</tr>
<tr>
<td>191</td>
<td>0.001397</td>
</tr>
<tr>
<td>192</td>
<td>0.001383</td>
</tr>
<tr>
<td>193</td>
<td>0.001370</td>
</tr>
<tr>
<td>194</td>
<td>0.001357</td>
</tr>
<tr>
<td>195</td>
<td>0.001344</td>
</tr>
<tr>
<td>196</td>
<td>0.001332</td>
</tr>
<tr>
<td>197</td>
<td>0.001320</td>
</tr>
<tr>
<td>198</td>
<td>0.001308</td>
</tr>
<tr>
<td>199</td>
<td>0.001296</td>
</tr>
<tr>
<td>200</td>
<td>0.001285</td>
</tr>
<tr>
<td>201</td>
<td>0.001274</td>
</tr>
<tr>
<td>202</td>
<td>0.001264</td>
</tr>
<tr>
<td>203</td>
<td>0.001254</td>
</tr>
<tr>
<td>204</td>
<td>0.001244</td>
</tr>
<tr>
<td>205</td>
<td>0.001235</td>
</tr>
<tr>
<td>206</td>
<td>0.001226</td>
</tr>
<tr>
<td>207</td>
<td>0.001218</td>
</tr>
<tr>
<td>208</td>
<td>0.001210</td>
</tr>
<tr>
<td>209</td>
<td>0.001203</td>
</tr>
<tr>
<td>210</td>
<td>0.001196</td>
</tr>
<tr>
<td>211</td>
<td>0.001190</td>
</tr>
<tr>
<td>212</td>
<td>0.001184</td>
</tr>
<tr>
<td>213</td>
<td>0.001179</td>
</tr>
<tr>
<td>214</td>
<td>0.001174</td>
</tr>
<tr>
<td>215</td>
<td>0.001169</td>
</tr>
<tr>
<td>216</td>
<td>0.001164</td>
</tr>
<tr>
<td>217</td>
<td>0.001159</td>
</tr>
<tr>
<td>218</td>
<td>0.001155</td>
</tr>
<tr>
<td>219</td>
<td>0.001151</td>
</tr>
<tr>
<td>220</td>
<td>0.001147</td>
</tr>
<tr>
<td>221</td>
<td>0.001143</td>
</tr>
<tr>
<td>222</td>
<td>0.001139</td>
</tr>
<tr>
<td>223</td>
<td>0.001136</td>
</tr>
<tr>
<td>224</td>
<td>0.001133</td>
</tr>
<tr>
<td>225</td>
<td>0.001130</td>
</tr>
<tr>
<td>226</td>
<td>0.001127</td>
</tr>
<tr>
<td>227</td>
<td>0.001124</td>
</tr>
<tr>
<td>228</td>
<td>0.001121</td>
</tr>
<tr>
<td>229</td>
<td>0.001118</td>
</tr>
</tbody>
</table>

Absolute Error, SN = 4130 (Continued)
| L9D: 0.001262 |  |  |
| L9B: 0.001245 |  |  |
| 19F: 0.001201 |  |  |
| lAO: 0.001413 |  |  |
| lAI: 0.001700 |  |  |
| lAZ: 0.001361 |  |  |
| lA3: 0.001321 |  |  |
| lA4: 0.001181 |  |  |
| lA5: 0.000872 |  |  |
| lA6: 0.001086 |  |  |
| lA7: 0.001060 |  |  |
| lA8: 0.001155 |  |  |
| lA9: 0.001338 |  |  |
| lAA: 0.001204 |  |  |
| lAB: 0.001230 |  |  |
| lAC: 0.001210 |  |  |
| lAD: 0.000971 |  |  |
| lAE: 0.001083 |  |  |
| lAF: 0.001274 |  |  |
| lBG: 0.001211 |  |  |
| lBH: 0.001133 |  |  |
| lBJ: 0.001069 |  |  |
| lB3: 0.001095 |  |  |
| lB4: 0.001065 |  |  |
| lB5: 0.001081 |  |  |
| lB6: 0.001124 |  |  |
| lB7: 0.001079 |  |  |
| lB8: 0.001040 |  |  |
| lB9: 0.001081 |  |  |
| lBA: 0.001183 |  |  |
| lBB: 0.001297 |  |  |
| lBC: 0.001124 |  |  |
| lBD: 0.001006 |  |  |
| lBE: 0.001046 |  |  |
| lBF: 0.001061 |  |  |
| lCC: 0.002475 |  |  |
| lCD: 0.002356 |  |  |
| lCE: 0.002530 |  |  |
| lCF: 0.002457 |  |  |
| lCG: 0.002712 |  |  |
| lCH: 0.002415 |  |  |
| lCI: 0.002579 |  |  |
| lCJ: 0.002436 |  |  |
| lCK: 0.002796 |  |  |
| lCL: 0.002388 |  |  |
| lCM: 0.002368 |  |  |
| lCN: 0.002426 |  |  |
| lCO: 0.002661 |  |  |
| lCP: 0.002462 |  |  |
| lCQ: 0.002497 |  |  |
| lCR: 0.002396 |  |  |
| lCS: 0.002617 |  |  |
| lCT: 0.002399 |  |  |
| lCU: 0.002533 |  |  |
| lCV: 0.002453 |  |  |
| lCX: 0.002423 |  |  |
| lCY: 0.002490 |  |  |
| lCZ: 0.002606 |  |  |

**Absolute Error, SN = 4130 (Continued)**

6-234
| ID9: | 0.002351: |
| IDA: | 0.002439: |
| IDB: | 0.002382: |
| IDC: | 0.002462: |
| IDD: | 0.002376: |
| IDE: | 0.002443: |
| IDE: | 0.002531: |
| IDB: | 0.002583: |
| IG1: | 0.002638: |
| IG2: | 0.002371: |
| IG3: | 0.002043: |
| IG4: | 0.002350: |
| IG5: | 0.002166: |
| IG6: | 0.002051: |
| IGF: | 0.002363: |
| IG8: | 0.002455: |
| IGH: | 0.002002: |
| IGA: | 0.002299: |
| IGB: | 0.002146: |
| IGC: | 0.002279: |
| IGD: | 0.002072: |
| IGE: | 0.001960: |
| IGF: | 0.002221: |
| IGA: | 0.002314: |
| IGJ: | 0.001940: |
| IGR: | 0.002086: |
| IGS: | 0.002310: |
| IGT: | 0.002180: |
| IGH: | 0.002075: |
| IGJ: | 0.002065: |
| IGL: | 0.002267: |
| IGM: | 0.002187: |
| IGP: | 0.002002: |
| IGR: | 0.002210: |
| IGH: | 0.002133: |
| IGL: | 0.002156: |
| IGA: | 0.001937: |
| IGJ: | 0.002097: |
| IGH: | 0.001409: |
| IGP: | 0.001699: |
| IGH: | 0.000107: |
| IGR: | 0.001905: |
| IGH: | 0.001557: |
| IGP: | 0.001650: |
| IGH: | 0.001661: |
| IGR: | 0.001683: |
| IGH: | 0.001595: |
| IGP: | 0.001535: |
| IGR: | 0.001179: |
| IGH: | 0.001610: |
| IGP: | 0.001454: |
| IGH: | 0.000170: |
| IGR: | 0.001262: |
| IGH: | 0.000129: |
| IGP: | 0.000993: |
| IGH: | 0.000405: |
| IGR: | 0.000107: |
| IGH: | 0.000168: |
| IGP: | 0.000193: |
| IGR: | 0.000420: |

**Absolute Error, SN = 4130 (Continued)**
<table>
<thead>
<tr>
<th>(x_i)</th>
<th>(y_i)</th>
<th>(z_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(215)</td>
<td>(0.001162)</td>
<td>(_)</td>
</tr>
<tr>
<td>(216)</td>
<td>(0.001323)</td>
<td>(_)</td>
</tr>
<tr>
<td>(217)</td>
<td>(0.001268)</td>
<td>(_)</td>
</tr>
<tr>
<td>(218)</td>
<td>(0.001296)</td>
<td>(_)</td>
</tr>
<tr>
<td>(219)</td>
<td>(0.001417)</td>
<td>(_)</td>
</tr>
<tr>
<td>(21A)</td>
<td>(0.001036)</td>
<td>(_)</td>
</tr>
<tr>
<td>(21B)</td>
<td>(0.001170)</td>
<td>(_)</td>
</tr>
<tr>
<td>(21C)</td>
<td>(0.001551)</td>
<td>(_)</td>
</tr>
<tr>
<td>(21D)</td>
<td>(0.001045)</td>
<td>(_)</td>
</tr>
<tr>
<td>(21E)</td>
<td>(0.001216)</td>
<td>(_)</td>
</tr>
<tr>
<td>(21F)</td>
<td>(0.000666)</td>
<td>(_)</td>
</tr>
<tr>
<td>(220)</td>
<td>(0.001304)</td>
<td>(_)</td>
</tr>
<tr>
<td>(221)</td>
<td>(0.000988)</td>
<td>(_)</td>
</tr>
<tr>
<td>(222)</td>
<td>(0.001207)</td>
<td>(_)</td>
</tr>
<tr>
<td>(223)</td>
<td>(0.001066)</td>
<td>(_)</td>
</tr>
<tr>
<td>(224)</td>
<td>(0.001079)</td>
<td>(_)</td>
</tr>
<tr>
<td>(225)</td>
<td>(0.001029)</td>
<td>(_)</td>
</tr>
<tr>
<td>(226)</td>
<td>(0.000971)</td>
<td>(_)</td>
</tr>
<tr>
<td>(227)</td>
<td>(0.000968)</td>
<td>(_)</td>
</tr>
<tr>
<td>(228)</td>
<td>(0.001203)</td>
<td>(_)</td>
</tr>
<tr>
<td>(229)</td>
<td>(0.000949)</td>
<td>(_)</td>
</tr>
<tr>
<td>(23A)</td>
<td>(0.001026)</td>
<td>(_)</td>
</tr>
<tr>
<td>(23B)</td>
<td>(0.001051)</td>
<td>(_)</td>
</tr>
<tr>
<td>(23C)</td>
<td>(0.001118)</td>
<td>(_)</td>
</tr>
<tr>
<td>(23D)</td>
<td>(0.000687)</td>
<td>(_)</td>
</tr>
<tr>
<td>(23E)</td>
<td>(0.001149)</td>
<td>(_)</td>
</tr>
<tr>
<td>(23F)</td>
<td>(0.000738)</td>
<td>(_)</td>
</tr>
<tr>
<td>(240)</td>
<td>(0.001214)</td>
<td>(_)</td>
</tr>
<tr>
<td>(241)</td>
<td>(0.000920)</td>
<td>(_)</td>
</tr>
<tr>
<td>(242)</td>
<td>(0.001203)</td>
<td>(_)</td>
</tr>
<tr>
<td>(243)</td>
<td>(0.000978)</td>
<td>(_)</td>
</tr>
<tr>
<td>(244)</td>
<td>(0.001203)</td>
<td>(_)</td>
</tr>
<tr>
<td>(245)</td>
<td>(0.001001)</td>
<td>(_)</td>
</tr>
<tr>
<td>(246)</td>
<td>(0.001003)</td>
<td>(_)</td>
</tr>
<tr>
<td>(247)</td>
<td>(0.001053)</td>
<td>(_)</td>
</tr>
<tr>
<td>(248)</td>
<td>(0.001235)</td>
<td>(_)</td>
</tr>
<tr>
<td>(249)</td>
<td>(0.000705)</td>
<td>(_)</td>
</tr>
<tr>
<td>(24A)</td>
<td>(0.001066)</td>
<td>(_)</td>
</tr>
<tr>
<td>(24B)</td>
<td>(0.000924)</td>
<td>(_)</td>
</tr>
<tr>
<td>(24C)</td>
<td>(0.001087)</td>
<td>(_)</td>
</tr>
<tr>
<td>(24D)</td>
<td>(0.001000)</td>
<td>(_)</td>
</tr>
<tr>
<td>(24E)</td>
<td>(0.001006)</td>
<td>(_)</td>
</tr>
<tr>
<td>(24F)</td>
<td>(0.000785)</td>
<td>(_)</td>
</tr>
<tr>
<td>(250)</td>
<td>(0.002137)</td>
<td>(_)</td>
</tr>
</tbody>
</table>

Absolute Error, \(SN = 4130\) (Continued)
| 251: | 0.002013: |  |
| 252: | 0.002259: |  |
| 253: | 0.002068: |  |
| 254: | 0.002370: |  |
| 255: | 0.002213: |  |
| 256: | 0.002314: |  |
| 257: | 0.002207: |  |
| 258: | 0.002259: |  |
| 259: | 0.002090: |  |
| 25A: | 0.001956: |  |
| 25B: | 0.002095: |  |
| 25C: | 0.002377: |  |
| 25D: | 0.002066: |  |
| 25E: | 0.002090: |  |
| 25F: | 0.001972: |  |
| 260: | 0.001237: |  |
| 261: | 0.001608: |  |
| 262: | 0.002022: |  |
| 263: | 0.001944: |  |
| 264: | 0.002053: |  |
| 265: | 0.001856: |  |
| 266: | 0.002042: |  |
| 267: | 0.001940: |  |
| 268: | 0.002039: |  |
| 269: | 0.001762: |  |
| 26A: | 0.001820: |  |
| 26B: | 0.001773: |  |
| 26C: | 0.001850: |  |
| 26D: | 0.001685: |  |
| 26E: | 0.001910: |  |
| 26F: | 0.001794: |  |
| 270: | 0.001748: |  |
| 271: | 0.001653: |  |
| 272: | 0.001632: |  |
| 273: | 0.001540: |  |
| 274: | 0.001677: |  |
| 275: | 0.001356: |  |
| 276: | 0.001582: |  |
| 277: | 0.001630: |  |
| 278: | 0.001505: |  |
| 279: | 0.001403: |  |
| 27A: | 0.001464: |  |
| 27B: | 0.001402: |  |
| 27C: | 0.001620: |  |
| 27D: | 0.001196: |  |
| 27E: | 0.001437: |  |
| 27F: | 0.001276: |  |
| 280: | 0.011913: |  |
| 281: | 0.001950: |  |
| 282: | 0.002095: |  |
| 283: | 0.001620: |  |
| 284: | 0.000996: |  |
| 285: | 0.001850: |  |
| 286: | 0.001951: |  |
| 287: | 0.001836: |  |
| 288: | 0.001726: |  |
| 289: | 0.001690: |  |
| 28A: | 0.001743: |  |
| 28B: | 0.001775: |  |
| 28C: | 0.001551: |  |

**Absolute Error, SN = 4130 (Continued)**
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>280:</td>
<td>0.001670</td>
<td>288:</td>
<td>0.001599</td>
</tr>
<tr>
<td>280:</td>
<td>0.001670</td>
<td>288:</td>
<td>0.001599</td>
</tr>
<tr>
<td>28F:</td>
<td>0.001536</td>
<td>290:</td>
<td>0.001558</td>
</tr>
<tr>
<td>291:</td>
<td>0.001423</td>
<td>292:</td>
<td>0.001437</td>
</tr>
<tr>
<td>293:</td>
<td>0.001255</td>
<td>294:</td>
<td>0.001423</td>
</tr>
<tr>
<td>295:</td>
<td>0.001151</td>
<td>296:</td>
<td>0.001336</td>
</tr>
<tr>
<td>297:</td>
<td>0.001131</td>
<td>298:</td>
<td>0.001060</td>
</tr>
<tr>
<td>299:</td>
<td>0.001125</td>
<td>29B:</td>
<td>0.001134</td>
</tr>
<tr>
<td>29C:</td>
<td>0.001209</td>
<td>29D:</td>
<td>0.000856</td>
</tr>
<tr>
<td>29E:</td>
<td>0.001095</td>
<td>29F:</td>
<td>0.000790</td>
</tr>
<tr>
<td>2AO:</td>
<td>0.000988</td>
<td>2A1:</td>
<td>0.000393</td>
</tr>
<tr>
<td>2A2:</td>
<td>0.001122</td>
<td>2A3:</td>
<td>0.000913</td>
</tr>
<tr>
<td>2A4:</td>
<td>0.000971</td>
<td>2A5:</td>
<td>0.000710</td>
</tr>
<tr>
<td>2A6:</td>
<td>0.000879</td>
<td>2A7:</td>
<td>0.000807</td>
</tr>
<tr>
<td>2A8:</td>
<td>0.001102</td>
<td>2A9:</td>
<td>0.000720</td>
</tr>
<tr>
<td>2AA:</td>
<td>0.000620</td>
<td>2AB:</td>
<td>0.000799</td>
</tr>
<tr>
<td>2AC:</td>
<td>0.000991</td>
<td>2AD:</td>
<td>0.000727</td>
</tr>
<tr>
<td>2AE:</td>
<td>0.000604</td>
<td>2AF:</td>
<td>0.000683</td>
</tr>
<tr>
<td>2B0:</td>
<td>0.000713</td>
<td>2B1:</td>
<td>0.000782</td>
</tr>
<tr>
<td>2B2:</td>
<td>0.000601</td>
<td>2B3:</td>
<td>0.000704</td>
</tr>
<tr>
<td>2B4:</td>
<td>0.000647</td>
<td>2B5:</td>
<td>0.000815</td>
</tr>
<tr>
<td>2B6:</td>
<td>0.000605</td>
<td>2B7:</td>
<td>0.000166</td>
</tr>
<tr>
<td>2B8:</td>
<td>0.000600</td>
<td>2B9:</td>
<td>0.000764</td>
</tr>
<tr>
<td>2BA:</td>
<td>0.000661</td>
<td>2BB:</td>
<td>0.000781</td>
</tr>
<tr>
<td>2BC:</td>
<td>0.000904</td>
<td>2BD:</td>
<td>0.000707</td>
</tr>
<tr>
<td>2BE:</td>
<td>0.000763</td>
<td>2BF:</td>
<td>0.000844</td>
</tr>
<tr>
<td>2C0:</td>
<td>0.002248</td>
<td>2C1:</td>
<td>0.001968</td>
</tr>
<tr>
<td>2C2:</td>
<td>0.002117</td>
<td>2C3:</td>
<td>0.002005</td>
</tr>
<tr>
<td>2C4:</td>
<td>0.002275</td>
<td>2C5:</td>
<td>0.002183</td>
</tr>
<tr>
<td>2C6:</td>
<td>0.002092</td>
<td>2C7:</td>
<td>0.002171</td>
</tr>
<tr>
<td>2C8:</td>
<td>0.002366</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Absolute Error, SN = 4130 (Continued)
Absolute Error, SN = 4130 (Continued)

2C9: 0.002105
2C8: 0.002047
2CB: 0.002142
2CC: 0.002308
2CD: 0.002226
2CE: 0.002106
2CF: 0.001931
2D0: 0.002290
2D1: 0.001963
2D2: 0.002106
2D3: 0.002014
2D4: 0.002136
2D5: 0.001849
2D6: 0.002152
2D7: 0.002205
2D8: 0.002087
2D9: 0.001866
2DA: 0.002304
2DB: 0.002224
2DC: 0.002308
2DD: 0.001769
2DE: 0.001555
2DF: 0.002034
2D0: 0.001801
2D1: 0.001780
2D2: 0.001813
2D3: 0.001724
2D4: 0.001537
2D5: 0.001622
2D6: 0.001797
2D7: 0.001799
2D8: 0.001720
2D9: 0.001537
2DA: 0.001715
2DB: 0.001385
2DC: 0.001667
2DD: 0.001664
2DE: 0.001508
2DF: 0.001373
2D0: 0.001408
2D1: 0.001379
2D2: 0.001508
2D3: 0.001325
2D4: 0.001385
2D5: 0.001225
2D6: 0.001381
2D7: 0.001301
2D8: 0.001168
2D9: 0.001136
2DA: 0.001032
2DB: 0.000957
2DC: 0.001102
2DD: 0.001008
2DE: 0.000999
2DF: 0.001571
300: 0.001494
301: 0.001278
302: 0.001463
303: 0.001298
304: 0.001282

270365-81
| 305 | 0.001267 |
| 306 | 0.001317 |
| 307 | 0.001154 |
| 308 | 0.001373 |
| 309 | 0.001001 |
| 30A | 0.001208 |
| 30B | 0.001134 |
| 30C | 0.001258 |
| 30D | 0.001135 |
| 30E | 0.001166 |
| 30F | 0.000971 |
| 310 | 0.001021 |
| 311 | 0.000689 |
| 312 | 0.000990 |
| 313 | 0.000657 |
| 314 | 0.000944 |
| 315 | 0.000651 |
| 316 | 0.000794 |
| 317 | 0.000744 |
| 318 | 0.000790 |
| 319 | 0.000702 |
| 31A | 0.000724 |
| 31B | 0.000613 |
| 31C | 0.000623 |
| 31D | 0.000691 |
| 31E | 0.000789 |
| 31F | -0.000670 |
| 320 | -0.000695 |
| 321 | -0.000923 |
| 322 | -0.000764 |
| 323 | -0.000845 |
| 324 | -0.000707 |
| 325 | -0.001111 |
| 326 | -0.000776 |
| 327 | -0.000991 |
| 328 | -0.000893 |
| 329 | -0.001089 |
| 32A | -0.000888 |
| 32B | -0.001001 |
| 32C | 0.001505 |
| 32D | 0.001350 |
| 32E | 0.001438 |
| 32F | 0.001358 |
| 330 | 0.001612 |
| 331 | 0.001368 |
| 332 | 0.001645 |
| 333 | 0.001482 |
| 334 | 0.001753 |
| 335 | 0.001664 |
| 336 | 0.001732 |
| 337 | 0.001582 |
| 338 | 0.001661 |
| 339 | 0.001472 |
| 33A | 0.001472 |
| 33B | 0.001522 |
| 33C | 0.001702 |
| 33D | 0.001371 |
| 33E | 0.001545 |
| 33F | 0.001281 |
| 340 | 0.002960 |

Absolute Error, SN = 4130 (Continued)
Absolute Error, \( SN = 4130 \) (Continued)

6-241
<table>
<thead>
<tr>
<th>SN</th>
<th>Absolute Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>370</td>
<td>0.001815</td>
</tr>
<tr>
<td>371</td>
<td>0.001669</td>
</tr>
<tr>
<td>372</td>
<td>0.002200</td>
</tr>
<tr>
<td>380</td>
<td>0.002064</td>
</tr>
<tr>
<td>381</td>
<td>0.001764</td>
</tr>
<tr>
<td>382</td>
<td>0.001910</td>
</tr>
<tr>
<td>383</td>
<td>0.001945</td>
</tr>
<tr>
<td>384</td>
<td>0.001913</td>
</tr>
<tr>
<td>385</td>
<td>0.001866</td>
</tr>
<tr>
<td>386</td>
<td>0.001809</td>
</tr>
<tr>
<td>387</td>
<td>0.001800</td>
</tr>
<tr>
<td>388</td>
<td>0.001779</td>
</tr>
<tr>
<td>389</td>
<td>0.001454</td>
</tr>
<tr>
<td>390</td>
<td>0.001504</td>
</tr>
<tr>
<td>391</td>
<td>0.001477</td>
</tr>
<tr>
<td>392</td>
<td>0.001469</td>
</tr>
<tr>
<td>393</td>
<td>0.001268</td>
</tr>
<tr>
<td>394</td>
<td>0.001256</td>
</tr>
<tr>
<td>395</td>
<td>0.001255</td>
</tr>
<tr>
<td>396</td>
<td>0.001087</td>
</tr>
<tr>
<td>397</td>
<td>0.001265</td>
</tr>
<tr>
<td>398</td>
<td>0.001421</td>
</tr>
<tr>
<td>399</td>
<td>0.001169</td>
</tr>
<tr>
<td>400</td>
<td>0.001269</td>
</tr>
<tr>
<td>401</td>
<td>0.001245</td>
</tr>
<tr>
<td>402</td>
<td>0.001440</td>
</tr>
<tr>
<td>403</td>
<td>0.001153</td>
</tr>
<tr>
<td>404</td>
<td>0.001462</td>
</tr>
<tr>
<td>405</td>
<td>0.001260</td>
</tr>
<tr>
<td>406</td>
<td>0.001363</td>
</tr>
<tr>
<td>407</td>
<td>0.001145</td>
</tr>
<tr>
<td>408</td>
<td>0.001221</td>
</tr>
<tr>
<td>409</td>
<td>0.001155</td>
</tr>
<tr>
<td>410</td>
<td>0.001452</td>
</tr>
<tr>
<td>411</td>
<td>0.001302</td>
</tr>
<tr>
<td>412</td>
<td>0.001138</td>
</tr>
<tr>
<td>413</td>
<td>0.001079</td>
</tr>
<tr>
<td>414</td>
<td>0.001378</td>
</tr>
<tr>
<td>415</td>
<td>0.001043</td>
</tr>
<tr>
<td>416</td>
<td>0.001145</td>
</tr>
<tr>
<td>417</td>
<td>0.001207</td>
</tr>
<tr>
<td>418</td>
<td>0.001161</td>
</tr>
<tr>
<td>419</td>
<td>0.001133</td>
</tr>
<tr>
<td>420</td>
<td>0.001137</td>
</tr>
<tr>
<td>421</td>
<td>0.001175</td>
</tr>
<tr>
<td>422</td>
<td>0.001159</td>
</tr>
<tr>
<td>423</td>
<td>0.000767</td>
</tr>
<tr>
<td>424</td>
<td>0.000927</td>
</tr>
<tr>
<td>425</td>
<td>0.000863</td>
</tr>
<tr>
<td>426</td>
<td>0.001127</td>
</tr>
<tr>
<td>427</td>
<td>0.000784</td>
</tr>
<tr>
<td>428</td>
<td>0.001002</td>
</tr>
<tr>
<td>429</td>
<td>0.001058</td>
</tr>
<tr>
<td>430</td>
<td>0.000907</td>
</tr>
</tbody>
</table>

Absolute Error, SN = 4130 (Continued)
| JBF | 0.001226: |
| JB : 0.001963: |
| JC : 0.001554: |
| JC : 0.001804: |
| JD : 0.002170: |
| JCE : 0.001396: |
| JCF : 0.001904: |
| JCG : 0.002087: |
| JCH : 0.002183: |
| JCI : 0.008204: |
| JCV : 0.002163: |
| JCW : 0.002036: |
| JCP : 0.002131: |
| JCR : 0.002017: |
| JCT : 0.001939: |
| JCU : 0.001809: |
| JCV : 0.001920: |
| JDW : 0.001776: |
| JDX : 0.002066: |
| JDP : 0.001764: |
| JDR : 0.001874: |
| JDS : 0.001861: |
| JDA : 0.001942: |
| JDC : 0.001800: |
| JDF : 0.001838: |
| JDG : 0.001993: |
| JDH : 0.001739: |
| JDI : 0.001712: |
| JDS : 0.001616: |
| JDV : 0.001576: |
| JDC : 0.001812: |
| JDD : 0.001652: |
| JD : 0.001872: |
| JDS : 0.001730: |
| JDT : 0.001546: |
| JDU : 0.001693: |
| JDA : 0.001857: |
| JDB : 0.001638: |
| JDC : 0.001738: |
| JDE : 0.001581: |
| JDF : 0.001579: |
| JDG : 0.001780: |
| JDS : 0.001451: |
| JDU : 0.001342: |
| JDA : 0.001439: |
| JDC : 0.001508: |
| JDD : 0.001463: |
| JDE : 0.001341: |
| JDF : 0.001340: |
| JDG : 0.001373: |
| JDS : 0.001106: |
| JDU : 0.001320: |
| JDA : 0.001046: |
| JDE : 0.001254: |
| JDF : 0.000000: |

**Absolute Error, SN = 4130 (Continued)**

6-243
Non. Lin. Error, SN = 4130

<table>
<thead>
<tr>
<th>Value</th>
<th>+ 0 -</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.00037</td>
<td></td>
<td>0.0037</td>
</tr>
<tr>
<td>0: -0.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: -0.000297</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: -0.000256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: -0.000363</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: 0.000031</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: -0.000266</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: -0.000134</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: -0.000397</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: 0.000007</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9: -0.000386</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A: -0.000210</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B: -0.000256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C: 0.000075</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D: -0.000247</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E: -0.000199</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F: -0.000468</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11: -0.000330</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12: -0.000094</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13: -0.000320</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14: 0.000040</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15: -0.000409</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16: -0.000277</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17: -0.000349</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18: -0.000009</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19: -0.000270</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A: -0.000112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B: -0.000296</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C: 0.000124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D: -0.000281</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000132</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000067</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000137</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000162</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000039</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000215</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000104</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000313</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000085</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000056</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000143</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000083</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000061</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000276</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000290</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: -0.000078</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000323</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.000211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>35:</td>
<td>-0.000065</td>
<td></td>
</tr>
<tr>
<td>36:</td>
<td>0.000165</td>
<td></td>
</tr>
<tr>
<td>37:</td>
<td>-0.000106</td>
<td></td>
</tr>
<tr>
<td>38:</td>
<td>0.000409</td>
<td></td>
</tr>
<tr>
<td>39:</td>
<td>0.000186</td>
<td></td>
</tr>
<tr>
<td>3A:</td>
<td>0.000368</td>
<td></td>
</tr>
<tr>
<td>3B:</td>
<td>0.000287</td>
<td></td>
</tr>
<tr>
<td>3C:</td>
<td>0.000275</td>
<td></td>
</tr>
<tr>
<td>3D:</td>
<td>0.002147</td>
<td></td>
</tr>
<tr>
<td>3E:</td>
<td>0.001651</td>
<td></td>
</tr>
<tr>
<td>3F:</td>
<td>0.001963</td>
<td></td>
</tr>
<tr>
<td>40:</td>
<td>0.001784</td>
<td></td>
</tr>
<tr>
<td>41:</td>
<td>0.001994</td>
<td></td>
</tr>
<tr>
<td>42:</td>
<td>0.001478</td>
<td></td>
</tr>
<tr>
<td>43:</td>
<td>0.001922</td>
<td></td>
</tr>
<tr>
<td>44:</td>
<td>0.001617</td>
<td></td>
</tr>
<tr>
<td>45:</td>
<td>0.001910</td>
<td></td>
</tr>
<tr>
<td>46:</td>
<td>0.001616</td>
<td></td>
</tr>
<tr>
<td>47:</td>
<td>0.001833</td>
<td></td>
</tr>
<tr>
<td>48:</td>
<td>0.001621</td>
<td></td>
</tr>
<tr>
<td>49:</td>
<td>0.001872</td>
<td></td>
</tr>
<tr>
<td>4A:</td>
<td>0.001585</td>
<td></td>
</tr>
<tr>
<td>4B:</td>
<td>0.001771</td>
<td></td>
</tr>
<tr>
<td>4C:</td>
<td>0.001392</td>
<td></td>
</tr>
<tr>
<td>4D:</td>
<td>0.001909</td>
<td></td>
</tr>
<tr>
<td>4E:</td>
<td>0.001682</td>
<td></td>
</tr>
<tr>
<td>4F:</td>
<td>0.001571</td>
<td></td>
</tr>
<tr>
<td>50:</td>
<td>0.001498</td>
<td></td>
</tr>
<tr>
<td>51:</td>
<td>0.001755</td>
<td></td>
</tr>
<tr>
<td>52:</td>
<td>0.001485</td>
<td></td>
</tr>
<tr>
<td>53:</td>
<td>0.001594</td>
<td></td>
</tr>
<tr>
<td>54:</td>
<td>0.001455</td>
<td></td>
</tr>
<tr>
<td>55:</td>
<td>0.001641</td>
<td></td>
</tr>
<tr>
<td>56:</td>
<td>0.001362</td>
<td></td>
</tr>
<tr>
<td>57:</td>
<td>0.001604</td>
<td></td>
</tr>
<tr>
<td>58:</td>
<td>0.001469</td>
<td></td>
</tr>
<tr>
<td>59:</td>
<td>0.001650</td>
<td></td>
</tr>
<tr>
<td>5A:</td>
<td>0.001323</td>
<td></td>
</tr>
<tr>
<td>5B:</td>
<td>0.001556</td>
<td></td>
</tr>
<tr>
<td>5C:</td>
<td>0.000952</td>
<td></td>
</tr>
<tr>
<td>5D:</td>
<td>0.001437</td>
<td></td>
</tr>
<tr>
<td>5E:</td>
<td>0.001163</td>
<td></td>
</tr>
<tr>
<td>5F:</td>
<td>0.001365</td>
<td></td>
</tr>
<tr>
<td>60:</td>
<td>0.001156</td>
<td></td>
</tr>
<tr>
<td>61:</td>
<td>0.001275</td>
<td></td>
</tr>
<tr>
<td>62:</td>
<td>0.000971</td>
<td></td>
</tr>
<tr>
<td>63:</td>
<td>0.001141</td>
<td></td>
</tr>
<tr>
<td>64:</td>
<td>0.000923</td>
<td></td>
</tr>
<tr>
<td>65:</td>
<td>0.000980</td>
<td></td>
</tr>
<tr>
<td>66:</td>
<td>0.000803</td>
<td></td>
</tr>
<tr>
<td>67:</td>
<td>0.000797</td>
<td></td>
</tr>
<tr>
<td>68:</td>
<td>0.000806</td>
<td></td>
</tr>
<tr>
<td>69:</td>
<td>0.000928</td>
<td></td>
</tr>
<tr>
<td>6A:</td>
<td>0.000589</td>
<td></td>
</tr>
<tr>
<td>6B:</td>
<td>0.000793</td>
<td></td>
</tr>
<tr>
<td>6C:</td>
<td>0.000592</td>
<td></td>
</tr>
<tr>
<td>6D:</td>
<td>0.000790</td>
<td></td>
</tr>
</tbody>
</table>

Non. Lin. Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th></th>
<th>Non. Lin. Error, SN = 4130 (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>0.000442:</td>
</tr>
<tr>
<td>72</td>
<td>0.000576:</td>
</tr>
<tr>
<td>73</td>
<td>0.000537:</td>
</tr>
<tr>
<td>74</td>
<td>0.000616:</td>
</tr>
<tr>
<td>75</td>
<td>0.000216:</td>
</tr>
<tr>
<td>76</td>
<td>0.000493:</td>
</tr>
<tr>
<td>77</td>
<td>0.000393:</td>
</tr>
<tr>
<td>78</td>
<td>0.000330:</td>
</tr>
<tr>
<td>79</td>
<td>0.000111:</td>
</tr>
<tr>
<td>80</td>
<td>0.000691:</td>
</tr>
<tr>
<td>81</td>
<td>0.000447:</td>
</tr>
<tr>
<td>82</td>
<td>0.000588:</td>
</tr>
<tr>
<td>83</td>
<td>0.000434:</td>
</tr>
<tr>
<td>84</td>
<td>0.000566:</td>
</tr>
<tr>
<td>85</td>
<td>0.000265:</td>
</tr>
<tr>
<td>86</td>
<td>0.000397:</td>
</tr>
<tr>
<td>87</td>
<td>0.000157:</td>
</tr>
<tr>
<td>88</td>
<td>0.000396:</td>
</tr>
<tr>
<td>89</td>
<td>-0.000196:</td>
</tr>
<tr>
<td>90</td>
<td>0.000339:</td>
</tr>
<tr>
<td>91</td>
<td>-0.000021:</td>
</tr>
<tr>
<td>92</td>
<td>0.000166:</td>
</tr>
<tr>
<td>93</td>
<td>-0.000104:</td>
</tr>
<tr>
<td>94</td>
<td>-0.000163:</td>
</tr>
<tr>
<td>95</td>
<td>-0.000285:</td>
</tr>
<tr>
<td>96</td>
<td>0.000089:</td>
</tr>
<tr>
<td>97</td>
<td>-0.000055:</td>
</tr>
<tr>
<td>98</td>
<td>0.000057:</td>
</tr>
<tr>
<td>99</td>
<td>-0.000129:</td>
</tr>
<tr>
<td>100</td>
<td>0.000025:</td>
</tr>
<tr>
<td>101</td>
<td>-0.000194:</td>
</tr>
<tr>
<td>102</td>
<td>0.000046:</td>
</tr>
<tr>
<td>103</td>
<td>-0.000255:</td>
</tr>
<tr>
<td>104</td>
<td>0.000119:</td>
</tr>
<tr>
<td>105</td>
<td>-0.000445:</td>
</tr>
<tr>
<td>106</td>
<td>-0.000214:</td>
</tr>
<tr>
<td>107</td>
<td>0.000376:</td>
</tr>
<tr>
<td>108</td>
<td>0.000305:</td>
</tr>
<tr>
<td>109</td>
<td>-0.000650:</td>
</tr>
<tr>
<td>110</td>
<td>0.000467:</td>
</tr>
<tr>
<td>111</td>
<td>-0.000967:</td>
</tr>
<tr>
<td>112</td>
<td>-0.000481:</td>
</tr>
<tr>
<td>113</td>
<td>0.000030:</td>
</tr>
<tr>
<td>114</td>
<td>-0.000416:</td>
</tr>
<tr>
<td>115</td>
<td>0.000079:</td>
</tr>
<tr>
<td>116</td>
<td>0.000157:</td>
</tr>
<tr>
<td>117</td>
<td>0.000574:</td>
</tr>
<tr>
<td>118</td>
<td>0.000045:</td>
</tr>
<tr>
<td>119</td>
<td>0.000119:</td>
</tr>
<tr>
<td>120</td>
<td>0.000098:</td>
</tr>
<tr>
<td>121</td>
<td>0.000077:</td>
</tr>
<tr>
<td>122</td>
<td>0.000092:</td>
</tr>
<tr>
<td>123</td>
<td>-0.000768:</td>
</tr>
<tr>
<td>124</td>
<td>-0.000011:</td>
</tr>
<tr>
<td>125</td>
<td>0.000824:</td>
</tr>
</tbody>
</table>

270365-35
| AD  | -0.001097  | | |
| AE  | -0.000960  | | |
| AF  | -0.001154  | | |
| B0  | -0.000787  | | |
| B1  | -0.001021  | | |
| B2  | -0.000909  | | |
| B3  | -0.001024  | | |
| B4  | -0.001064  | | |
| B5  | -0.001152  | | |
| B6  | -0.001051  | | |
| B7  | -0.001199  | | |
| B8  | -0.001089  | | |
| B9  | -0.001260  | | |
| BA  | -0.001104  | | |
| BB  | -0.001284  | | |
| BC  | -0.001242  | | |
| BD  | -0.001376  | | |
| BE  | -0.001174  | | |
| BF  | -0.001735  | | |
| C0  | 0.000398   | | |
| C1  | 0.000097   | | |
| C2  | 0.000248   | | |
| C3  | 0.000109   | | |
| C4  | 0.000216   | | |
| C5  | -0.000054  | | |
| C6  | 0.000322   | | |
| C7  | -0.000018  | | |
| C8  | 0.000254   | | |
| C9  | 0.000018   | | |
| CA  | -0.000866  | | |
| CB  | 0.000005   | | |
| CC  | 0.000208   | | |
| CD  | -0.000113  | | |
| CE  | 0.000094   | | |
| CF  | -0.000093  | | |
| D0  | 0.000058   | | |
| D1  | -0.000191  | | |
| D2  | -0.000049  | | |
| D3  | -0.000445  | | |
| D4  | 0.000056   | | |
| D5  | -0.000306  | | |
| D6  | 0.000023   | | |
| D7  | -0.000145  | | |
| D8  | -0.000116  | | |
| D9  | -0.000231  | | |
| DA  | -0.000044  | | |
| DB  | -0.000158  | | |
| DC  | -0.000168  | | |
| DD  | -0.000455  | | |
| DE  | -0.000301  | | |
| DF  | -0.000633  | | |
| EG  | -0.000326  | | |
| EL  | -0.000830  | | |
| E2  | -0.000421  | | |
| E3  | -0.000605  | | |
| E4  | -0.000729  | | |
| E5  | -0.000709  | | |
| E6  | -0.000527  | | |
| E7  | -0.000572  | | |
| E8  | -0.000462  | | |

Non. Lin. Error, SN = 4130 (Continued)

6-247
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>E9</td>
<td>-0.000694</td>
</tr>
<tr>
<td>E10</td>
<td>-0.000557</td>
</tr>
<tr>
<td>E11</td>
<td>-0.000709</td>
</tr>
<tr>
<td>E12</td>
<td>-0.000540</td>
</tr>
<tr>
<td>E13</td>
<td>-0.000752</td>
</tr>
<tr>
<td>E14</td>
<td>-0.000667</td>
</tr>
<tr>
<td>E15</td>
<td>-0.000667</td>
</tr>
<tr>
<td>E16</td>
<td>-0.000728</td>
</tr>
<tr>
<td>F1</td>
<td>-0.009989</td>
</tr>
<tr>
<td>F2</td>
<td>-0.000706</td>
</tr>
<tr>
<td>F3</td>
<td>-0.000947</td>
</tr>
<tr>
<td>F4</td>
<td>-0.000668</td>
</tr>
<tr>
<td>F5</td>
<td>-0.001156</td>
</tr>
<tr>
<td>F6</td>
<td>-0.000807</td>
</tr>
<tr>
<td>F7</td>
<td>-0.001030</td>
</tr>
<tr>
<td>F8</td>
<td>-0.001200</td>
</tr>
<tr>
<td>F9</td>
<td>-0.001222</td>
</tr>
<tr>
<td>F10</td>
<td>-0.000956</td>
</tr>
<tr>
<td>F11</td>
<td>-0.001087</td>
</tr>
<tr>
<td>F12</td>
<td>-0.000663</td>
</tr>
<tr>
<td>F13</td>
<td>-0.000977</td>
</tr>
<tr>
<td>F14</td>
<td>-0.001857</td>
</tr>
<tr>
<td>F15</td>
<td>-0.000659</td>
</tr>
<tr>
<td>F16</td>
<td>-0.000843</td>
</tr>
<tr>
<td>F17</td>
<td>-0.000606</td>
</tr>
<tr>
<td>F18</td>
<td>-0.000628</td>
</tr>
<tr>
<td>F19</td>
<td>-0.000659</td>
</tr>
<tr>
<td>F20</td>
<td>-0.000982</td>
</tr>
<tr>
<td>F21</td>
<td>-0.000973</td>
</tr>
<tr>
<td>F22</td>
<td>-0.001042</td>
</tr>
<tr>
<td>F23</td>
<td>-0.000755</td>
</tr>
<tr>
<td>F24</td>
<td>-0.001040</td>
</tr>
<tr>
<td>F25</td>
<td>-0.000904</td>
</tr>
<tr>
<td>F26</td>
<td>-0.000902</td>
</tr>
<tr>
<td>F27</td>
<td>-0.000912</td>
</tr>
<tr>
<td>F28</td>
<td>-0.001173</td>
</tr>
<tr>
<td>F29</td>
<td>-0.001030</td>
</tr>
<tr>
<td>F30</td>
<td>-0.001382</td>
</tr>
<tr>
<td>F31</td>
<td>-0.001240</td>
</tr>
<tr>
<td>F32</td>
<td>-0.001386</td>
</tr>
<tr>
<td>F33</td>
<td>-0.001093</td>
</tr>
<tr>
<td>F34</td>
<td>-0.001244</td>
</tr>
<tr>
<td>F35</td>
<td>-0.001240</td>
</tr>
<tr>
<td>F36</td>
<td>-0.001503</td>
</tr>
<tr>
<td>F37</td>
<td>-0.001328</td>
</tr>
<tr>
<td>F38</td>
<td>-0.001400</td>
</tr>
<tr>
<td>F39</td>
<td>-0.001521</td>
</tr>
<tr>
<td>F40</td>
<td>-0.001494</td>
</tr>
<tr>
<td>F41</td>
<td>-0.001614</td>
</tr>
<tr>
<td>F42</td>
<td>-0.001565</td>
</tr>
<tr>
<td>F43</td>
<td>-0.001814</td>
</tr>
<tr>
<td>F44</td>
<td>-0.001621</td>
</tr>
<tr>
<td>F45</td>
<td>-0.002076</td>
</tr>
<tr>
<td>F46</td>
<td>-0.001541</td>
</tr>
<tr>
<td>F47</td>
<td>-0.001841</td>
</tr>
<tr>
<td>F48</td>
<td>-0.001758</td>
</tr>
<tr>
<td>F49</td>
<td>-0.001790</td>
</tr>
<tr>
<td>F50</td>
<td>-0.001644</td>
</tr>
<tr>
<td>F51</td>
<td>-0.001604</td>
</tr>
</tbody>
</table>

Non. Lin. Error, SN = 4130 (Continued)
| 125:  | -0.002140: |
| 126:  | -0.001946: |
| 127:  | -0.001975: |
| 128:  | -0.001772: |
| 129:  | -0.001967: |
| 130:  | -0.001774: |
| 131:  | -0.001888: |
| 132:  | -0.002048: |
| 133:  | -0.002140: |
| 134:  | -0.001900: |
| 135:  | -0.002249: |
| 136:  | -0.002054: |
| 137:  | -0.002233: |
| 138:  | -0.002039: |
| 139:  | -0.002342: |
| 140:  | -0.002083: |
| 141:  | -0.002333: |
| 142:  | -0.001896: |
| 143:  | -0.002125: |
| 144:  | -0.002177: |
| 145:  | -0.002168: |
| 146:  | -0.002097: |
| 147:  | -0.002142: |
| 148:  | -0.002018: |
| 149:  | -0.002490: |
| 150:  | -0.000666: |
| 151:  | -0.000700: |
| 152:  | -0.000560: |
| 153:  | -0.000692: |
| 154:  | -0.000493: |
| 155:  | -0.000724: |
| 156:  | -0.000607: |
| 157:  | -0.000659: |
| 158:  | -0.000441: |
| 159:  | -0.000712: |
| 160:  | -0.000578: |
| 161:  | -0.000517: |
| 162:  | -0.000527: |
| 163:  | -0.000753: |
| 164:  | -0.000650: |
| 165:  | -0.000857: |
| 166:  | -0.000688: |
| 167:  | -0.000880: |
| 168:  | -0.000746: |
| 169:  | -0.000917: |
| 170:  | -0.000747: |
| 171:  | -0.000986: |
| 172:  | -0.000929: |
| 173:  | -0.000931: |
| 174:  | -0.000731: |
| 175:  | -0.001008: |
| 176:  | -0.000998: |
| 177:  | -0.000972: |
| 178:  | -0.000867: |
| 179:  | -0.001075: |
| 180:  | -0.000914: |
| 181:  | -0.001140: |
| 182:  | -0.001002: |

**Non. Lin. Error, SN = 4130** (Continued)
<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>161</td>
<td>-0.001273</td>
</tr>
<tr>
<td>162</td>
<td>-0.001057</td>
</tr>
<tr>
<td>163</td>
<td>-0.001275</td>
</tr>
<tr>
<td>164</td>
<td>-0.001046</td>
</tr>
<tr>
<td>165</td>
<td>-0.001502</td>
</tr>
<tr>
<td>166</td>
<td>-0.001155</td>
</tr>
<tr>
<td>167</td>
<td>-0.001274</td>
</tr>
<tr>
<td>168</td>
<td>-0.001100</td>
</tr>
<tr>
<td>169</td>
<td>-0.001259</td>
</tr>
<tr>
<td>16A</td>
<td>-0.000968</td>
</tr>
<tr>
<td>16B</td>
<td>-0.001205</td>
</tr>
<tr>
<td>16C</td>
<td>-0.001170</td>
</tr>
<tr>
<td>16D</td>
<td>-0.001449</td>
</tr>
<tr>
<td>16E</td>
<td>-0.001375</td>
</tr>
<tr>
<td>16F</td>
<td>-0.001347</td>
</tr>
<tr>
<td>170</td>
<td>-0.001278</td>
</tr>
<tr>
<td>171</td>
<td>-0.001557</td>
</tr>
<tr>
<td>172</td>
<td>-0.001365</td>
</tr>
<tr>
<td>173</td>
<td>-0.001430</td>
</tr>
<tr>
<td>174</td>
<td>-0.001328</td>
</tr>
<tr>
<td>175</td>
<td>-0.001520</td>
</tr>
<tr>
<td>176</td>
<td>-0.001455</td>
</tr>
<tr>
<td>177</td>
<td>-0.001460</td>
</tr>
<tr>
<td>178</td>
<td>-0.001315</td>
</tr>
<tr>
<td>179</td>
<td>-0.001617</td>
</tr>
<tr>
<td>17A</td>
<td>-0.001338</td>
</tr>
<tr>
<td>17B</td>
<td>-0.001464</td>
</tr>
<tr>
<td>17C</td>
<td>-0.001486</td>
</tr>
<tr>
<td>17D</td>
<td>-0.001679</td>
</tr>
<tr>
<td>17E</td>
<td>-0.001500</td>
</tr>
<tr>
<td>17F</td>
<td>-0.001611</td>
</tr>
<tr>
<td>180</td>
<td>-0.001090</td>
</tr>
<tr>
<td>181</td>
<td>-0.001379</td>
</tr>
<tr>
<td>182</td>
<td>-0.001306</td>
</tr>
<tr>
<td>183</td>
<td>-0.001366</td>
</tr>
<tr>
<td>184</td>
<td>-0.001230</td>
</tr>
<tr>
<td>185</td>
<td>-0.001470</td>
</tr>
<tr>
<td>186</td>
<td>-0.001377</td>
</tr>
<tr>
<td>187</td>
<td>-0.001460</td>
</tr>
<tr>
<td>188</td>
<td>-0.001309</td>
</tr>
<tr>
<td>189</td>
<td>-0.001553</td>
</tr>
<tr>
<td>18A</td>
<td>-0.001370</td>
</tr>
<tr>
<td>18B</td>
<td>-0.001493</td>
</tr>
<tr>
<td>18C</td>
<td>-0.001353</td>
</tr>
<tr>
<td>18D</td>
<td>-0.001682</td>
</tr>
<tr>
<td>18E</td>
<td>-0.001502</td>
</tr>
<tr>
<td>18F</td>
<td>-0.001670</td>
</tr>
<tr>
<td>190</td>
<td>-0.001229</td>
</tr>
<tr>
<td>191</td>
<td>-0.001624</td>
</tr>
<tr>
<td>192</td>
<td>-0.001671</td>
</tr>
<tr>
<td>193</td>
<td>-0.001674</td>
</tr>
<tr>
<td>194</td>
<td>-0.001672</td>
</tr>
<tr>
<td>195</td>
<td>-0.001841</td>
</tr>
<tr>
<td>196</td>
<td>-0.001669</td>
</tr>
<tr>
<td>197</td>
<td>-0.002024</td>
</tr>
<tr>
<td>198</td>
<td>-0.001466</td>
</tr>
<tr>
<td>199</td>
<td>-0.001871</td>
</tr>
<tr>
<td>19A</td>
<td>-0.001688</td>
</tr>
<tr>
<td>19B</td>
<td>-0.001782</td>
</tr>
<tr>
<td>19C</td>
<td>-0.001516</td>
</tr>
</tbody>
</table>

Non. Lin. Error, SN = 4130 (Continued)
| iAO | -0.001845 |
| iAF | -0.001807 |
| 1B0 | -0.001822 |
| 1B1 | -0.002051 |
| 1B2 | -0.001916 |
| 1B3 | -0.001991 |
| 1B4 | -0.001973 |
| 1B5 | -0.002108 |
| 1B6 | -0.002067 |
| 1B7 | -0.002013 |
| 1B8 | -0.002053 |
| 1B9 | -0.002113 |
| 1BA | -0.001913 |
| 1BB | -0.001950 |
| 1BC | -0.001974 |
| 1BD | -0.002144 |
| 1BE | -0.002055 |
| 1BF | -0.002041 |
| 1CO | -0.000629 |
| 1C1 | -0.000707 |
| 1C2 | -0.000569 |
| 1C3 | -0.000701 |
| 1C4 | -0.000497 |
| 1C5 | -0.000746 |
| 1C6 | -0.000533 |
| 1C7 | -0.000677 |
| 1C8 | -0.000419 |
| 1C9 | -0.000728 |
| 1CA | -0.000699 |
| 1CB | -0.000643 |
| 1CC | -0.000599 |
| 1CD | -0.000759 |
| 1CE | -0.000676 |
| 1CF | -0.000678 |
| 1DD | -0.000508 |
| 1D1 | -0.000770 |
| 1D2 | -0.000675 |
| 1D3 | -0.000726 |
| 1D4 | -0.000558 |
| 1D5 | -0.000768 |
| 1D6 | -0.000760 |
| 1D7 | -0.000645 |
| 1D8 | -0.000580 |

Non. Lin. Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>109</td>
<td>-0.000836</td>
</tr>
<tr>
<td>10A</td>
<td>-0.000750</td>
</tr>
<tr>
<td>10B</td>
<td>-0.000758</td>
</tr>
<tr>
<td>10C</td>
<td>-0.000715</td>
</tr>
<tr>
<td>10D</td>
<td>-0.000816</td>
</tr>
<tr>
<td>10E</td>
<td>-0.000701</td>
</tr>
<tr>
<td>10F</td>
<td>-0.000714</td>
</tr>
<tr>
<td>10G</td>
<td>-0.000663</td>
</tr>
<tr>
<td>10H</td>
<td>-0.001060</td>
</tr>
<tr>
<td>10I</td>
<td>-0.000828</td>
</tr>
<tr>
<td>10J</td>
<td>-0.001107</td>
</tr>
<tr>
<td>10K</td>
<td>-0.000802</td>
</tr>
<tr>
<td>10L</td>
<td>-0.001037</td>
</tr>
<tr>
<td>10M</td>
<td>-0.000803</td>
</tr>
<tr>
<td>10N</td>
<td>-0.000843</td>
</tr>
<tr>
<td>10O</td>
<td>-0.000702</td>
</tr>
<tr>
<td>10P</td>
<td>-0.001156</td>
</tr>
<tr>
<td>10Q</td>
<td>-0.000861</td>
</tr>
<tr>
<td>10R</td>
<td>-0.000965</td>
</tr>
<tr>
<td>10S</td>
<td>-0.000933</td>
</tr>
<tr>
<td>10T</td>
<td>-0.001142</td>
</tr>
<tr>
<td>10U</td>
<td>-0.001205</td>
</tr>
<tr>
<td>10V</td>
<td>-0.000995</td>
</tr>
<tr>
<td>10W</td>
<td>-0.000954</td>
</tr>
<tr>
<td>10X</td>
<td>-0.001179</td>
</tr>
<tr>
<td>10Y</td>
<td>-0.001084</td>
</tr>
<tr>
<td>10Z</td>
<td>-0.001061</td>
</tr>
<tr>
<td>11A</td>
<td>-0.001035</td>
</tr>
<tr>
<td>11B</td>
<td>-0.001099</td>
</tr>
<tr>
<td>11C</td>
<td>-0.001111</td>
</tr>
<tr>
<td>11D</td>
<td>-0.000960</td>
</tr>
<tr>
<td>11E</td>
<td>-0.000991</td>
</tr>
<tr>
<td>11F</td>
<td>-0.001170</td>
</tr>
<tr>
<td>11G</td>
<td>-0.001061</td>
</tr>
<tr>
<td>11H</td>
<td>-0.001099</td>
</tr>
<tr>
<td>11I</td>
<td>-0.001182</td>
</tr>
<tr>
<td>11J</td>
<td>-0.001248</td>
</tr>
<tr>
<td>11K</td>
<td>-0.001157</td>
</tr>
<tr>
<td>11L</td>
<td>-0.001828</td>
</tr>
<tr>
<td>11M</td>
<td>-0.001360</td>
</tr>
<tr>
<td>11N</td>
<td>-0.001583</td>
</tr>
<tr>
<td>11O</td>
<td>-0.001386</td>
</tr>
<tr>
<td>11P</td>
<td>-0.001636</td>
</tr>
<tr>
<td>11Q</td>
<td>-0.001536</td>
</tr>
<tr>
<td>11R</td>
<td>-0.001584</td>
</tr>
<tr>
<td>11S</td>
<td>-0.001514</td>
</tr>
<tr>
<td>11T</td>
<td>-0.001703</td>
</tr>
<tr>
<td>11U</td>
<td>-0.001714</td>
</tr>
<tr>
<td>11V</td>
<td>-0.002021</td>
</tr>
<tr>
<td>11W</td>
<td>-0.001592</td>
</tr>
<tr>
<td>11X</td>
<td>-0.001799</td>
</tr>
<tr>
<td>11Y</td>
<td>-0.001884</td>
</tr>
<tr>
<td>11Z</td>
<td>-0.001994</td>
</tr>
<tr>
<td>20A</td>
<td>-0.002028</td>
</tr>
<tr>
<td>20B</td>
<td>-0.002225</td>
</tr>
<tr>
<td>20C</td>
<td>-0.001805</td>
</tr>
<tr>
<td>20D</td>
<td>-0.002137</td>
</tr>
<tr>
<td>20E</td>
<td>-0.001994</td>
</tr>
<tr>
<td>20F</td>
<td>-0.002071</td>
</tr>
<tr>
<td>20G</td>
<td>-0.001795</td>
</tr>
</tbody>
</table>

Non. Lin. Error, SN = 4130 (Continued)
Non. Lin. Error, SN = 4130 (Continued)
Non. Lin. Error, $SN = 4130$ (Continued)
| 28D: | -0.001855: | #   |
| 28E: | -0.001778: | #   |
| 28F: | -0.001942: | #   |
| 290: | -0.001871: | #   |
| 291: | -0.002008: | #   |
| 292: | -0.001945: | #   |
| 293: | -0.002128: | #   |
| 294: | -0.001962: | #   |
| 295: | -0.002235: | #   |
| 296: | -0.002101: | #   |
| 297: | -0.002178: | #   |
| 298: | -0.002132: | #   |
| 299: | -0.002368: | #   |
| 29A: | -0.002433: | #   |
| 29B: | -0.002360: | #   |
| 29C: | -0.002236: | #   |
| 29D: | -0.002541: | #   |
| 29E: | -0.002403: | #   |
| 29F: | -0.002609: | #   |
| 2A0: | -0.002413: | #   |
| 2A1: | -0.002563: | #   |
| 2A2: | -0.002381: | #   |
| 2A3: | -0.002542: | #   |
| 2A4: | -0.002435: | #   |
| 2A5: | -0.002697: | #   |
| 2A6: | -0.002530: | #   |
| 2A7: | -0.002653: | #   |
| 2A8: | -0.002459: | #   |
| 2A9: | -0.002742: | #   |
| 2AA: | -0.002866: | #   |
| 2AB: | -0.002666: | #   |
| 2AC: | -0.002525: | #   |
| 2AD: | -0.002741: | #   |
| 2AF: | -0.002737: | #   |
| 2B0: | -0.002709: | #   |
| 2B1: | -0.003031: | #   |
| 2B2: | -0.002823: | #   |
| 2B3: | -0.002906: | #   |
| 2B4: | -0.002780: | #   |
| 2B5: | -0.003019: | #   |
| 2B6: | -0.002941: | #   |
| 2B7: | -0.002923: | #   |
| 2B8: | -0.002794: | #   |
| 2B9: | -0.002973: | #   |
| 2BA: | -0.002872: | #   |
| 2BB: | -0.002943: | #   |
| 2BC: | -0.002584: | #   |
| 2BD: | -0.002832: | #   |
| 2BE: | -0.002777: | #   |
| 2BF: | -0.002698: | #   |
| 2C0: | -0.001295: | #   |
| 2C1: | -0.001557: | #   |
| 2C2: | -0.001429: | #   |
| 2C3: | -0.001542: | #   |
| 2C4: | -0.001274: | #   |
| 2C5: | -0.001417: | #   |
| 2C6: | -0.001409: | #   |
| 2C7: | -0.001382: | #   |
| 2C8: | -0.001138: | #   |

Non. Lin. Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2C4</td>
<td>-0.001450</td>
</tr>
<tr>
<td>2CA</td>
<td>-0.001609</td>
</tr>
<tr>
<td>2CB</td>
<td>-0.001366</td>
</tr>
<tr>
<td>2CC</td>
<td>-0.001201</td>
</tr>
<tr>
<td>2CD</td>
<td>-0.001385</td>
</tr>
<tr>
<td>2CE</td>
<td>-0.001503</td>
</tr>
<tr>
<td>2CF</td>
<td>-0.001334</td>
</tr>
<tr>
<td>2D0</td>
<td>-0.001166</td>
</tr>
<tr>
<td>2D1</td>
<td>-0.001369</td>
</tr>
<tr>
<td>2D2</td>
<td>-0.001438</td>
</tr>
<tr>
<td>2D3</td>
<td>-0.001660</td>
</tr>
<tr>
<td>2DA</td>
<td>-0.001550</td>
</tr>
<tr>
<td>2DB</td>
<td>-0.001060</td>
</tr>
<tr>
<td>2DC</td>
<td>-0.001166</td>
</tr>
<tr>
<td>2DD</td>
<td>-0.001503</td>
</tr>
<tr>
<td>2DE</td>
<td>-0.001799</td>
</tr>
<tr>
<td>2DF</td>
<td>-0.001725</td>
</tr>
<tr>
<td>2E0</td>
<td>-0.001766</td>
</tr>
<tr>
<td>2E1</td>
<td>-0.001550</td>
</tr>
<tr>
<td>2E2</td>
<td>-0.001700</td>
</tr>
<tr>
<td>2E3</td>
<td>-0.001685</td>
</tr>
<tr>
<td>2E4</td>
<td>-0.001766</td>
</tr>
<tr>
<td>2E5</td>
<td>-0.001747</td>
</tr>
<tr>
<td>2E6</td>
<td>-0.001796</td>
</tr>
<tr>
<td>2E7</td>
<td>-0.001776</td>
</tr>
<tr>
<td>2E8</td>
<td>-0.001776</td>
</tr>
<tr>
<td>2E9</td>
<td>-0.002011</td>
</tr>
<tr>
<td>2EA</td>
<td>-0.001834</td>
</tr>
<tr>
<td>2EB</td>
<td>-0.001615</td>
</tr>
<tr>
<td>2EC</td>
<td>-0.002132</td>
</tr>
<tr>
<td>2ED</td>
<td>-0.002177</td>
</tr>
<tr>
<td>2EE</td>
<td>-0.002236</td>
</tr>
<tr>
<td>2EF</td>
<td>-0.002284</td>
</tr>
<tr>
<td>2F0</td>
<td>-0.002315</td>
</tr>
<tr>
<td>2F1</td>
<td>-0.002449</td>
</tr>
<tr>
<td>2F2</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2F3</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2F4</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2F5</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2F6</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2F7</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2F8</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2F9</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2FA</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2FB</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2FC</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2FD</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2FE</td>
<td>-0.002533</td>
</tr>
<tr>
<td>2FF</td>
<td>-0.002533</td>
</tr>
<tr>
<td>300</td>
<td>-0.001994</td>
</tr>
<tr>
<td>301</td>
<td>-0.002301</td>
</tr>
<tr>
<td>302</td>
<td>-0.002168</td>
</tr>
<tr>
<td>303</td>
<td>-0.002284</td>
</tr>
<tr>
<td>304</td>
<td>-0.002251</td>
</tr>
</tbody>
</table>

Non. Lin. Error, SN = 4130 (Continued)
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>305:</td>
<td>-0.002117</td>
</tr>
<tr>
<td>306:</td>
<td>-0.002269</td>
</tr>
<tr>
<td>307:</td>
<td>-0.002483</td>
</tr>
<tr>
<td>308:</td>
<td>-0.002265</td>
</tr>
<tr>
<td>309:</td>
<td>-0.002569</td>
</tr>
<tr>
<td>30A:</td>
<td>-0.002383</td>
</tr>
<tr>
<td>30B:</td>
<td>-0.002508</td>
</tr>
<tr>
<td>30C:</td>
<td>-0.002336</td>
</tr>
<tr>
<td>30D:</td>
<td>-0.002560</td>
</tr>
<tr>
<td>30E:</td>
<td>-0.002428</td>
</tr>
<tr>
<td>30F:</td>
<td>-0.002589</td>
</tr>
<tr>
<td>310:</td>
<td>-0.002383</td>
</tr>
<tr>
<td>311:</td>
<td>-0.002508</td>
</tr>
<tr>
<td>312:</td>
<td>-0.002336</td>
</tr>
<tr>
<td>313:</td>
<td>-0.002560</td>
</tr>
<tr>
<td>314:</td>
<td>-0.002428</td>
</tr>
<tr>
<td>315:</td>
<td>-0.002589</td>
</tr>
<tr>
<td>316:</td>
<td>-0.002383</td>
</tr>
<tr>
<td>317:</td>
<td>-0.002508</td>
</tr>
<tr>
<td>318:</td>
<td>-0.002336</td>
</tr>
<tr>
<td>319:</td>
<td>-0.002560</td>
</tr>
<tr>
<td>31A:</td>
<td>-0.002428</td>
</tr>
<tr>
<td>31B:</td>
<td>-0.002589</td>
</tr>
<tr>
<td>31C:</td>
<td>-0.002383</td>
</tr>
<tr>
<td>31D:</td>
<td>-0.002508</td>
</tr>
<tr>
<td>31E:</td>
<td>-0.002336</td>
</tr>
<tr>
<td>31F:</td>
<td>-0.002560</td>
</tr>
<tr>
<td>320:</td>
<td>-0.002428</td>
</tr>
<tr>
<td>321:</td>
<td>-0.002589</td>
</tr>
<tr>
<td>322:</td>
<td>-0.002383</td>
</tr>
<tr>
<td>323:</td>
<td>-0.002508</td>
</tr>
<tr>
<td>324:</td>
<td>-0.002336</td>
</tr>
<tr>
<td>325:</td>
<td>-0.002560</td>
</tr>
<tr>
<td>326:</td>
<td>-0.002428</td>
</tr>
<tr>
<td>327:</td>
<td>-0.002589</td>
</tr>
<tr>
<td>328:</td>
<td>-0.002383</td>
</tr>
<tr>
<td>329:</td>
<td>-0.002508</td>
</tr>
<tr>
<td>32A:</td>
<td>-0.002336</td>
</tr>
<tr>
<td>32B:</td>
<td>-0.002560</td>
</tr>
<tr>
<td>32C:</td>
<td>-0.002428</td>
</tr>
<tr>
<td>32D:</td>
<td>-0.002589</td>
</tr>
<tr>
<td>32E:</td>
<td>-0.002383</td>
</tr>
<tr>
<td>32F:</td>
<td>-0.002508</td>
</tr>
</tbody>
</table>

**Non. Lin. Error, SN = 4130 (Continued)**

6-257
Non. Lin. Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Location</th>
<th>Non. Lin. Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>378:</td>
<td>0.001979</td>
</tr>
<tr>
<td>379:</td>
<td>0.002106</td>
</tr>
<tr>
<td>37F:</td>
<td>0.001957</td>
</tr>
<tr>
<td>380:</td>
<td>0.001784</td>
</tr>
<tr>
<td>381:</td>
<td>0.002085</td>
</tr>
<tr>
<td>382:</td>
<td>0.001810</td>
</tr>
<tr>
<td>383:</td>
<td>0.001907</td>
</tr>
<tr>
<td>384:</td>
<td>0.001990</td>
</tr>
<tr>
<td>385:</td>
<td>0.001989</td>
</tr>
<tr>
<td>386:</td>
<td>0.001867</td>
</tr>
<tr>
<td>387:</td>
<td>0.001957</td>
</tr>
<tr>
<td>388:</td>
<td>0.002029</td>
</tr>
<tr>
<td>389:</td>
<td>0.002556</td>
</tr>
<tr>
<td>38A:</td>
<td>0.002177</td>
</tr>
<tr>
<td>38B:</td>
<td>0.002285</td>
</tr>
<tr>
<td>38C:</td>
<td>0.002294</td>
</tr>
<tr>
<td>38D:</td>
<td>0.002249</td>
</tr>
<tr>
<td>38E:</td>
<td>0.002204</td>
</tr>
<tr>
<td>38F:</td>
<td>0.002499</td>
</tr>
<tr>
<td>390:</td>
<td>0.002201</td>
</tr>
<tr>
<td>391:</td>
<td>0.002774</td>
</tr>
<tr>
<td>392:</td>
<td>0.002398</td>
</tr>
<tr>
<td>393:</td>
<td>0.002591</td>
</tr>
<tr>
<td>394:</td>
<td>0.002325</td>
</tr>
<tr>
<td>395:</td>
<td>0.002570</td>
</tr>
<tr>
<td>396:</td>
<td>0.002590</td>
</tr>
<tr>
<td>397:</td>
<td>0.002513</td>
</tr>
<tr>
<td>398:</td>
<td>0.002609</td>
</tr>
<tr>
<td>399:</td>
<td>0.002662</td>
</tr>
<tr>
<td>39A:</td>
<td>0.002513</td>
</tr>
<tr>
<td>39B:</td>
<td>0.002588</td>
</tr>
<tr>
<td>39C:</td>
<td>0.002345</td>
</tr>
<tr>
<td>39D:</td>
<td>0.002633</td>
</tr>
<tr>
<td>39E:</td>
<td>0.002405</td>
</tr>
<tr>
<td>39F:</td>
<td>0.002579</td>
</tr>
<tr>
<td>3A0:</td>
<td>0.002427</td>
</tr>
<tr>
<td>3A1:</td>
<td>0.002646</td>
</tr>
<tr>
<td>3A2:</td>
<td>0.002572</td>
</tr>
<tr>
<td>3A3:</td>
<td>0.002639</td>
</tr>
<tr>
<td>3A4:</td>
<td>0.002393</td>
</tr>
<tr>
<td>3A5:</td>
<td>0.003494</td>
</tr>
<tr>
<td>3A6:</td>
<td>0.002609</td>
</tr>
<tr>
<td>3A7:</td>
<td>0.002570</td>
</tr>
<tr>
<td>3A8:</td>
<td>0.002152</td>
</tr>
<tr>
<td>3A9:</td>
<td>0.002809</td>
</tr>
<tr>
<td>3AA:</td>
<td>0.002658</td>
</tr>
<tr>
<td>3AB:</td>
<td>0.002690</td>
</tr>
<tr>
<td>3AC:</td>
<td>0.002645</td>
</tr>
<tr>
<td>3AD:</td>
<td>0.002724</td>
</tr>
<tr>
<td>3AE:</td>
<td>0.002721</td>
</tr>
<tr>
<td>3AF:</td>
<td>0.002685</td>
</tr>
<tr>
<td>3B0:</td>
<td>0.002752</td>
</tr>
<tr>
<td>3B1:</td>
<td>0.003015</td>
</tr>
<tr>
<td>3B2:</td>
<td>0.002837</td>
</tr>
<tr>
<td>3B3:</td>
<td>0.002932</td>
</tr>
<tr>
<td>3B4:</td>
<td>0.002609</td>
</tr>
<tr>
<td>3B5:</td>
<td>0.003034</td>
</tr>
<tr>
<td>3B6:</td>
<td>0.002817</td>
</tr>
<tr>
<td>3B7:</td>
<td>0.002812</td>
</tr>
<tr>
<td>3B8:</td>
<td>0.002965</td>
</tr>
</tbody>
</table>

Non. Lin. Error, SN = 4130 (Continued)
Non. Lin. Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00000000</td>
<td>10</td>
<td>0.000467</td>
</tr>
<tr>
<td>1</td>
<td>-0.000297</td>
<td>11</td>
<td>-0.000329</td>
</tr>
<tr>
<td>2</td>
<td>0.000041</td>
<td>12</td>
<td>0.000235</td>
</tr>
<tr>
<td>3</td>
<td>-0.000067</td>
<td>13</td>
<td>-0.000226</td>
</tr>
<tr>
<td>4</td>
<td>0.0000371</td>
<td>14</td>
<td>0.000361</td>
</tr>
<tr>
<td>5</td>
<td>-0.000297</td>
<td>15</td>
<td>-0.000450</td>
</tr>
<tr>
<td>6</td>
<td>0.0000151</td>
<td>16</td>
<td>0.000131</td>
</tr>
<tr>
<td>7</td>
<td>-0.000263</td>
<td>17</td>
<td>-0.000071</td>
</tr>
<tr>
<td>8</td>
<td>0.0000405</td>
<td>18</td>
<td>0.0000339</td>
</tr>
<tr>
<td>9</td>
<td>-0.000394</td>
<td>19</td>
<td>-0.000261</td>
</tr>
<tr>
<td>A</td>
<td>0.000175</td>
<td>1A</td>
<td>0.000158</td>
</tr>
<tr>
<td>B</td>
<td>-0.000045</td>
<td>1B</td>
<td>-0.000184</td>
</tr>
<tr>
<td>C</td>
<td>0.0000331</td>
<td>1C</td>
<td>0.000171</td>
</tr>
<tr>
<td>D</td>
<td>-0.000323</td>
<td>1D</td>
<td>-0.0000356</td>
</tr>
<tr>
<td>E</td>
<td>0.000048</td>
<td>1E</td>
<td>0.000146</td>
</tr>
<tr>
<td>F</td>
<td>-0.000269</td>
<td>1F</td>
<td>-0.000475</td>
</tr>
<tr>
<td>10</td>
<td>0.0000467</td>
<td>20</td>
<td>0.0000745</td>
</tr>
<tr>
<td>11</td>
<td>-0.000329</td>
<td>21</td>
<td>-0.000300</td>
</tr>
<tr>
<td>12</td>
<td>0.0000235</td>
<td>22</td>
<td>0.0000122</td>
</tr>
<tr>
<td>13</td>
<td>-0.000226</td>
<td>23</td>
<td>-0.000175</td>
</tr>
<tr>
<td>14</td>
<td>0.000361</td>
<td>24</td>
<td>0.0000319</td>
</tr>
<tr>
<td>15</td>
<td>-0.000450</td>
<td>25</td>
<td>-0.000076</td>
</tr>
<tr>
<td>16</td>
<td>0.000131</td>
<td>26</td>
<td>-0.000027</td>
</tr>
<tr>
<td>17</td>
<td>-0.000071</td>
<td>27</td>
<td>-0.000314</td>
</tr>
<tr>
<td>18</td>
<td>0.0000339</td>
<td>28</td>
<td>0.0000436</td>
</tr>
<tr>
<td>19</td>
<td>-0.000261</td>
<td>29</td>
<td>-0.000208</td>
</tr>
<tr>
<td>1A</td>
<td>0.000158</td>
<td>2A</td>
<td>0.000141</td>
</tr>
<tr>
<td>1B</td>
<td>-0.000184</td>
<td>2B</td>
<td>-0.000200</td>
</tr>
<tr>
<td>1C</td>
<td>0.000171</td>
<td>2C</td>
<td>0.0000361</td>
</tr>
<tr>
<td>1D</td>
<td>-0.0000356</td>
<td>2D</td>
<td>-0.000301</td>
</tr>
<tr>
<td>1E</td>
<td>0.000146</td>
<td>2E</td>
<td>0.000144</td>
</tr>
<tr>
<td>1F</td>
<td>-0.000475</td>
<td>2F</td>
<td>-0.000038</td>
</tr>
<tr>
<td>20</td>
<td>0.0000745</td>
<td>30</td>
<td>0.000567</td>
</tr>
<tr>
<td>21</td>
<td>-0.000300</td>
<td>31</td>
<td>-0.000282</td>
</tr>
<tr>
<td>22</td>
<td>0.0000122</td>
<td>32</td>
<td>0.000069</td>
</tr>
<tr>
<td>23</td>
<td>-0.000175</td>
<td>33</td>
<td>-0.000401</td>
</tr>
<tr>
<td>24</td>
<td>0.0000319</td>
<td>34</td>
<td>0.000534</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130
<p>| 35: | -0.000277: |
| 36: | 0.000231: |
| 37: | -0.000272: |
| 38: | 0.000516: |
| 39: | -0.000223: |
| 3A: | 0.000211: |
| 3B: | -0.000212: |
| 3C: | 0.000200: |
| 3D: | -0.000081: |
| 3E: | 0.000225: |
| 3F: | -0.000788: |
| 40: | 0.002423: |
| 41: | -0.000496: |
| 42: | 0.000331: |
| 43: | -0.000199: |
| 44: | 0.000210: |
| 45: | -0.000516: |
| 46: | 0.000643: |
| 47: | -0.000304: |
| 48: | 0.000292: |
| 49: | -0.000094: |
| 4A: | 0.000217: |
| 4B: | -0.000212: |
| 4C: | 0.000250: |
| 4D: | -0.000286: |
| 4E: | 0.000185: |
| 4F: | -0.000379: |
| 50: | 0.000508: |
| 51: | -0.000218: |
| 52: | 0.000111: |
| 53: | -0.000722: |
| 54: | 0.000256: |
| 55: | -0.000270: |
| 56: | 0.000109: |
| 57: | -0.000139: |
| 58: | 0.000185: |
| 59: | -0.000258: |
| 5A: | 0.000221: |
| 5B: | -0.000115: |
| 5C: | 0.000161: |
| 5D: | -0.000327: |
| 5E: | 0.000212: |
| 5F: | -0.000584: |
| 60: | 0.000485: |
| 61: | -0.000274: |
| 62: | 0.000201: |
| 63: | -0.000208: |
| 64: | 0.000118: |
| 65: | -0.000304: |
| 66: | 0.000170: |
| 67: | -0.000218: |
| 68: | 0.000056: |
| 69: | -0.000176: |
| 6A: | -0.000006: |
| 6B: | 0.000008: |
| 6C: | 0.000122: |
| 6D: | 0.000339: |
| 6E: | 0.000203: |
| 6F: | -0.000201: |
| 70: | 0.000206: |</p>
<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Value</th>
<th>Offset</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0.000000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>01</td>
<td>0.000013</td>
<td>0.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td>02</td>
<td>0.000008</td>
<td>0.0002</td>
<td>0.0002</td>
</tr>
<tr>
<td>03</td>
<td>0.000000</td>
<td>0.0003</td>
<td>0.0003</td>
</tr>
<tr>
<td>04</td>
<td>0.000000</td>
<td>0.0004</td>
<td>0.0004</td>
</tr>
<tr>
<td>05</td>
<td>0.000000</td>
<td>0.0005</td>
<td>0.0005</td>
</tr>
<tr>
<td>06</td>
<td>0.000000</td>
<td>0.0006</td>
<td>0.0006</td>
</tr>
<tr>
<td>07</td>
<td>0.000000</td>
<td>0.0007</td>
<td>0.0007</td>
</tr>
<tr>
<td>08</td>
<td>0.000000</td>
<td>0.0008</td>
<td>0.0008</td>
</tr>
<tr>
<td>09</td>
<td>0.000000</td>
<td>0.0009</td>
<td>0.0009</td>
</tr>
<tr>
<td>0A</td>
<td>0.000000</td>
<td>0.000A</td>
<td>0.000A</td>
</tr>
<tr>
<td>0B</td>
<td>0.000000</td>
<td>0.000B</td>
<td>0.000B</td>
</tr>
<tr>
<td>0C</td>
<td>0.000000</td>
<td>0.000C</td>
<td>0.000C</td>
</tr>
<tr>
<td>0D</td>
<td>0.000000</td>
<td>0.000D</td>
<td>0.000D</td>
</tr>
<tr>
<td>0E</td>
<td>0.000000</td>
<td>0.000E</td>
<td>0.000E</td>
</tr>
<tr>
<td>0F</td>
<td>0.000000</td>
<td>0.000F</td>
<td>0.000F</td>
</tr>
<tr>
<td>10</td>
<td>0.000000</td>
<td>0.0010</td>
<td>0.0010</td>
</tr>
<tr>
<td>11</td>
<td>0.000000</td>
<td>0.0011</td>
<td>0.0011</td>
</tr>
<tr>
<td>12</td>
<td>0.000000</td>
<td>0.0012</td>
<td>0.0012</td>
</tr>
<tr>
<td>13</td>
<td>0.000000</td>
<td>0.0013</td>
<td>0.0013</td>
</tr>
<tr>
<td>14</td>
<td>0.000000</td>
<td>0.0014</td>
<td>0.0014</td>
</tr>
<tr>
<td>15</td>
<td>0.000000</td>
<td>0.0015</td>
<td>0.0015</td>
</tr>
<tr>
<td>16</td>
<td>0.000000</td>
<td>0.0016</td>
<td>0.0016</td>
</tr>
<tr>
<td>17</td>
<td>0.000000</td>
<td>0.0017</td>
<td>0.0017</td>
</tr>
<tr>
<td>18</td>
<td>0.000000</td>
<td>0.0018</td>
<td>0.0018</td>
</tr>
<tr>
<td>19</td>
<td>0.000000</td>
<td>0.0019</td>
<td>0.0019</td>
</tr>
<tr>
<td>1A</td>
<td>0.000000</td>
<td>0.001A</td>
<td>0.001A</td>
</tr>
<tr>
<td>1B</td>
<td>0.000000</td>
<td>0.001B</td>
<td>0.001B</td>
</tr>
<tr>
<td>1C</td>
<td>0.000000</td>
<td>0.001C</td>
<td>0.001C</td>
</tr>
<tr>
<td>1D</td>
<td>0.000000</td>
<td>0.001D</td>
<td>0.001D</td>
</tr>
<tr>
<td>1E</td>
<td>0.000000</td>
<td>0.001E</td>
<td>0.001E</td>
</tr>
<tr>
<td>1F</td>
<td>0.000000</td>
<td>0.001F</td>
<td>0.001F</td>
</tr>
<tr>
<td>20</td>
<td>0.000000</td>
<td>0.0020</td>
<td>0.0020</td>
</tr>
<tr>
<td>21</td>
<td>0.000000</td>
<td>0.0021</td>
<td>0.0021</td>
</tr>
<tr>
<td>22</td>
<td>0.000000</td>
<td>0.0022</td>
<td>0.0022</td>
</tr>
<tr>
<td>23</td>
<td>0.000000</td>
<td>0.0023</td>
<td>0.0023</td>
</tr>
<tr>
<td>24</td>
<td>0.000000</td>
<td>0.0024</td>
<td>0.0024</td>
</tr>
<tr>
<td>25</td>
<td>0.000000</td>
<td>0.0025</td>
<td>0.0025</td>
</tr>
<tr>
<td>26</td>
<td>0.000000</td>
<td>0.0026</td>
<td>0.0026</td>
</tr>
<tr>
<td>27</td>
<td>0.000000</td>
<td>0.0027</td>
<td>0.0027</td>
</tr>
<tr>
<td>28</td>
<td>0.000000</td>
<td>0.0028</td>
<td>0.0028</td>
</tr>
<tr>
<td>29</td>
<td>0.000000</td>
<td>0.0029</td>
<td>0.0029</td>
</tr>
<tr>
<td>2A</td>
<td>0.000000</td>
<td>0.002A</td>
<td>0.002A</td>
</tr>
<tr>
<td>2B</td>
<td>0.000000</td>
<td>0.002B</td>
<td>0.002B</td>
</tr>
<tr>
<td>2C</td>
<td>0.000000</td>
<td>0.002C</td>
<td>0.002C</td>
</tr>
<tr>
<td>2D</td>
<td>0.000000</td>
<td>0.002D</td>
<td>0.002D</td>
</tr>
<tr>
<td>2E</td>
<td>0.000000</td>
<td>0.002E</td>
<td>0.002E</td>
</tr>
<tr>
<td>2F</td>
<td>0.000000</td>
<td>0.002F</td>
<td>0.002F</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0.00000</td>
</tr>
<tr>
<td>01</td>
<td>0.00017</td>
</tr>
<tr>
<td>02</td>
<td>0.00013</td>
</tr>
<tr>
<td>03</td>
<td>-0.00019</td>
</tr>
<tr>
<td>04</td>
<td>0.00037</td>
</tr>
<tr>
<td>05</td>
<td>-0.00037</td>
</tr>
<tr>
<td>06</td>
<td>0.00034</td>
</tr>
<tr>
<td>07</td>
<td>-0.00034</td>
</tr>
<tr>
<td>08</td>
<td>0.00027</td>
</tr>
<tr>
<td>09</td>
<td>-0.00027</td>
</tr>
<tr>
<td>0A</td>
<td>0.00010</td>
</tr>
<tr>
<td>0B</td>
<td>-0.00010</td>
</tr>
<tr>
<td>0C</td>
<td>0.00004</td>
</tr>
<tr>
<td>0D</td>
<td>-0.00013</td>
</tr>
<tr>
<td>0E</td>
<td>0.00020</td>
</tr>
<tr>
<td>0F</td>
<td>-0.00020</td>
</tr>
<tr>
<td>10</td>
<td>0.00020</td>
</tr>
<tr>
<td>11</td>
<td>-0.00020</td>
</tr>
<tr>
<td>12</td>
<td>0.00020</td>
</tr>
<tr>
<td>13</td>
<td>-0.00020</td>
</tr>
<tr>
<td>14</td>
<td>0.00020</td>
</tr>
<tr>
<td>15</td>
<td>-0.00020</td>
</tr>
<tr>
<td>16</td>
<td>0.00020</td>
</tr>
<tr>
<td>17</td>
<td>-0.00020</td>
</tr>
<tr>
<td>18</td>
<td>0.00020</td>
</tr>
<tr>
<td>19</td>
<td>-0.00020</td>
</tr>
<tr>
<td>1A</td>
<td>0.00020</td>
</tr>
<tr>
<td>1B</td>
<td>-0.00020</td>
</tr>
<tr>
<td>1C</td>
<td>0.00020</td>
</tr>
<tr>
<td>1D</td>
<td>-0.00020</td>
</tr>
<tr>
<td>1E</td>
<td>0.00020</td>
</tr>
<tr>
<td>1F</td>
<td>-0.00020</td>
</tr>
</tbody>
</table>

**DNL Error, SN = 4130 (Continued)**
DNL Error, SN = 4130 (Continued)
| 125:  | -0.000496:  |
| 126:  | 0.000193:   |
| 127:  | -0.000028:  |
| 128:  | 0.000202:   |
| 129:  | -0.000194:  |
| 130:  | 0.000048:   |
| 131:  | -0.001000:  |
| 132:  | 0.000168:   |
| 133:  | -0.000253:  |
| 134:  | 0.000188:   |
| 135:  | -0.000178:  |
| 136:  | 0.000193:   |
| 137:  | -0.000303:  |
| 138:  | 0.000259:   |
| 139:  | -0.000250:  |
| 140:  | 0.000250:   |
| 141:  | -0.000436:  |
| 142:  | 0.000220:   |
| 143:  | -0.000052:  |
| 144:  | 0.000008:   |
| 145:  | -0.000271:  |
| 146:  | 0.000245:   |
| 147:  | 0.0000123:  |
| 148:  | -0.000471:  |
| 149:  | 0.001823:   |
| 150:  | -0.000033:  |
| 151:  | 0.000139:   |
| 152:  | -0.000132:  |
| 153:  | 0.000199:   |
| 154:  | -0.000231:  |
| 155:  | 0.000116:   |
| 156:  | -0.000051:  |
| 157:  | 0.000217:   |
| 158:  | -0.000271:  |
| 159:  | 0.000134:   |
| 160:  | 0.000060:   |
| 161:  | -0.000010:  |
| 162:  | 0.000225:   |
| 163:  | 0.000102:   |
| 164:  | -0.000207:  |
| 165:  | 0.000168:   |
| 166:  | 0.000191:   |
| 167:  | -0.000133:  |
| 168:  | 0.000171:   |
| 169:  | -0.000170:  |
| 170:  | 0.000239:   |
| 171:  | 0.000056:   |
| 172:  | -0.000001:  |
| 173:  | 0.000199:   |
| 174:  | -0.000277:  |
| 175:  | 0.000110:   |
| 176:  | 0.000074:   |
| 177:  | -0.000104:  |
| 178:  | 0.000207:   |
| 179:  | 0.000160:   |
| 180:  | -0.000226:  |
| 181:  | 0.000138:   |

DNL Error, SN = 4130 (Continued)
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>161:</td>
<td>-0.000271</td>
</tr>
<tr>
<td>162:</td>
<td>0.000215</td>
</tr>
<tr>
<td>163:</td>
<td>-0.000217</td>
</tr>
<tr>
<td>164:</td>
<td>0.000226</td>
</tr>
<tr>
<td>165:</td>
<td>-0.000454</td>
</tr>
<tr>
<td>166:</td>
<td>0.000347</td>
</tr>
<tr>
<td>167:</td>
<td>-0.000119</td>
</tr>
<tr>
<td>168:</td>
<td>0.000173</td>
</tr>
<tr>
<td>169:</td>
<td>-0.000158</td>
</tr>
<tr>
<td>170:</td>
<td>0.000027</td>
</tr>
<tr>
<td>171:</td>
<td>-0.000279</td>
</tr>
<tr>
<td>172:</td>
<td>0.000191</td>
</tr>
<tr>
<td>173:</td>
<td>-0.000064</td>
</tr>
<tr>
<td>174:</td>
<td>0.000101</td>
</tr>
<tr>
<td>175:</td>
<td>-0.000192</td>
</tr>
<tr>
<td>176:</td>
<td>0.000065</td>
</tr>
<tr>
<td>177:</td>
<td>-0.000025</td>
</tr>
<tr>
<td>178:</td>
<td>0.000164</td>
</tr>
<tr>
<td>179:</td>
<td>-0.000301</td>
</tr>
<tr>
<td>180:</td>
<td>0.000278</td>
</tr>
<tr>
<td>181:</td>
<td>-0.000146</td>
</tr>
<tr>
<td>182:</td>
<td>0.000001</td>
</tr>
<tr>
<td>183:</td>
<td>-0.000193</td>
</tr>
<tr>
<td>184:</td>
<td>0.000178</td>
</tr>
<tr>
<td>185:</td>
<td>-0.000110</td>
</tr>
<tr>
<td>186:</td>
<td>0.000512</td>
</tr>
<tr>
<td>187:</td>
<td>-0.000281</td>
</tr>
<tr>
<td>188:</td>
<td>0.000073</td>
</tr>
<tr>
<td>189:</td>
<td>-0.000060</td>
</tr>
<tr>
<td>190:</td>
<td>0.000135</td>
</tr>
<tr>
<td>191:</td>
<td>-0.000267</td>
</tr>
<tr>
<td>192:</td>
<td>0.000100</td>
</tr>
<tr>
<td>193:</td>
<td>-0.000103</td>
</tr>
<tr>
<td>194:</td>
<td>0.000171</td>
</tr>
<tr>
<td>195:</td>
<td>-0.000244</td>
</tr>
<tr>
<td>196:</td>
<td>0.000174</td>
</tr>
<tr>
<td>197:</td>
<td>-0.000114</td>
</tr>
<tr>
<td>198:</td>
<td>0.000139</td>
</tr>
<tr>
<td>199:</td>
<td>-0.000329</td>
</tr>
<tr>
<td>200:</td>
<td>0.000180</td>
</tr>
<tr>
<td>201:</td>
<td>-0.000176</td>
</tr>
<tr>
<td>202:</td>
<td>0.000448</td>
</tr>
<tr>
<td>203:</td>
<td>-0.000395</td>
</tr>
<tr>
<td>204:</td>
<td>0.000046</td>
</tr>
<tr>
<td>205:</td>
<td>-0.000003</td>
</tr>
<tr>
<td>206:</td>
<td>0.000001</td>
</tr>
<tr>
<td>207:</td>
<td>-0.000168</td>
</tr>
<tr>
<td>208:</td>
<td>0.000171</td>
</tr>
<tr>
<td>209:</td>
<td>-0.000355</td>
</tr>
<tr>
<td>210:</td>
<td>0.000558</td>
</tr>
<tr>
<td>211:</td>
<td>-0.000405</td>
</tr>
<tr>
<td>212:</td>
<td>0.000182</td>
</tr>
<tr>
<td>213:</td>
<td>-0.000093</td>
</tr>
<tr>
<td>214:</td>
<td>0.000265</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130 (Continued)
| IA1: | -0.000244: | |
| IA2: | 0.000139: | |
| IA3: | -0.000091: | |
| IA4: | 0.000041: | |
| IA5: | -0.000210: | |
| IA6: | 0.000162: | |
| IA7: | -0.000057: | |
| IA8: | 0.000163: | |
| IA9: | 0.000158: | |
| IAA: | 0.000114: | |
| IAB: | 0.000224: | |
| IAC: | 0.000028: | |
| IAD: | -0.000240: | |
| IAE: | 0.000110: | |
| IAF: | 0.000189: | |
| IAG: | -0.000014: | |
| IAI: | -0.000229: | |
| IAJ: | 0.000134: | |
| IAK: | -0.000075: | |
| IAL: | 0.000181: | |
| IAM: | -0.000135: | |
| IAN: | 0.000041: | |
| IAO: | 0.000053: | |
| IAP: | -0.000040: | |
| IAQ: | 0.000060: | |
| IRA: | 0.000200: | |
| IRB: | -0.000037: | |
| IRC: | -0.00024: | |
| IRI: | -0.000269: | |
| IRS: | -0.000051: | |
| IRT: | 0.000168: | |
| IRU: | 0.000210: | |
| IRV: | -0.000007: | |
| IRW: | 0.000115: | |
| IXX: | 0.000064: | |

**DNL Error, SN = 4130 (Continued)**

6-268
<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>09</td>
<td>-0.000256</td>
</tr>
<tr>
<td>0A</td>
<td>0.000086</td>
</tr>
<tr>
<td>0B</td>
<td>-0.00008</td>
</tr>
<tr>
<td>0C</td>
<td>0.00042</td>
</tr>
<tr>
<td>0D</td>
<td>-0.00101</td>
</tr>
<tr>
<td>0E</td>
<td>0.00115</td>
</tr>
<tr>
<td>0F</td>
<td>-0.00013</td>
</tr>
<tr>
<td>10</td>
<td>0.00050</td>
</tr>
<tr>
<td>11</td>
<td>-0.00036</td>
</tr>
<tr>
<td>12</td>
<td>0.00023</td>
</tr>
<tr>
<td>13</td>
<td>-0.00026</td>
</tr>
<tr>
<td>14</td>
<td>0.00030</td>
</tr>
<tr>
<td>15</td>
<td>-0.0002</td>
</tr>
<tr>
<td>16</td>
<td>0.00023</td>
</tr>
<tr>
<td>17</td>
<td>-0.00004</td>
</tr>
<tr>
<td>18</td>
<td>0.00114</td>
</tr>
<tr>
<td>19</td>
<td>-0.00045</td>
</tr>
<tr>
<td>1A</td>
<td>0.00029</td>
</tr>
<tr>
<td>1B</td>
<td>-0.00014</td>
</tr>
<tr>
<td>1C</td>
<td>0.00031</td>
</tr>
<tr>
<td>1D</td>
<td>-0.00208</td>
</tr>
<tr>
<td>1E</td>
<td>-0.00063</td>
</tr>
<tr>
<td>1F</td>
<td>0.00029</td>
</tr>
<tr>
<td>20</td>
<td>0.00041</td>
</tr>
<tr>
<td>21</td>
<td>-0.00022</td>
</tr>
<tr>
<td>22</td>
<td>0.00094</td>
</tr>
<tr>
<td>23</td>
<td>0.00023</td>
</tr>
<tr>
<td>24</td>
<td>0.00025</td>
</tr>
<tr>
<td>25</td>
<td>-0.00064</td>
</tr>
<tr>
<td>26</td>
<td>0.00011</td>
</tr>
<tr>
<td>27</td>
<td>0.00015</td>
</tr>
<tr>
<td>28</td>
<td>0.00031</td>
</tr>
<tr>
<td>29</td>
<td>-0.00186</td>
</tr>
<tr>
<td>2A</td>
<td>0.00016</td>
</tr>
<tr>
<td>2B</td>
<td>-0.00038</td>
</tr>
<tr>
<td>2C</td>
<td>0.00073</td>
</tr>
<tr>
<td>2D</td>
<td>-0.00222</td>
</tr>
<tr>
<td>2E</td>
<td>0.00090</td>
</tr>
<tr>
<td>2F</td>
<td>-0.00067</td>
</tr>
<tr>
<td>40</td>
<td>0.00046</td>
</tr>
<tr>
<td>41</td>
<td>-0.00023</td>
</tr>
<tr>
<td>42</td>
<td>0.00096</td>
</tr>
<tr>
<td>43</td>
<td>-0.00024</td>
</tr>
<tr>
<td>44</td>
<td>0.00009</td>
</tr>
<tr>
<td>45</td>
<td>0.00048</td>
</tr>
<tr>
<td>46</td>
<td>0.00009</td>
</tr>
<tr>
<td>47</td>
<td>0.00070</td>
</tr>
<tr>
<td>48</td>
<td>0.00018</td>
</tr>
<tr>
<td>49</td>
<td>-0.00001</td>
</tr>
<tr>
<td>4A</td>
<td>0.00007</td>
</tr>
<tr>
<td>4B</td>
<td>-0.00036</td>
</tr>
<tr>
<td>4C</td>
<td>-0.00085</td>
</tr>
<tr>
<td>4D</td>
<td>-0.00019</td>
</tr>
<tr>
<td>4E</td>
<td>0.00003</td>
</tr>
<tr>
<td>4F</td>
<td>-0.00019</td>
</tr>
<tr>
<td>50</td>
<td>0.00042</td>
</tr>
<tr>
<td>51</td>
<td>0.00033</td>
</tr>
<tr>
<td>52</td>
<td>0.00014</td>
</tr>
<tr>
<td>53</td>
<td>0.00007</td>
</tr>
<tr>
<td>54</td>
<td>0.00023</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>215</td>
<td>-0.000309</td>
</tr>
<tr>
<td>216</td>
<td>0.000159</td>
</tr>
<tr>
<td>217</td>
<td>-0.000056</td>
</tr>
<tr>
<td>218</td>
<td>0.000026</td>
</tr>
<tr>
<td>219</td>
<td>0.000020</td>
</tr>
<tr>
<td>21A</td>
<td>0.000012</td>
</tr>
<tr>
<td>21B</td>
<td>0.000082</td>
</tr>
<tr>
<td>21C</td>
<td>0.000379</td>
</tr>
<tr>
<td>21D</td>
<td>-0.000407</td>
</tr>
<tr>
<td>21E</td>
<td>0.000199</td>
</tr>
<tr>
<td>21F</td>
<td>-0.000051</td>
</tr>
<tr>
<td>220</td>
<td>0.000536</td>
</tr>
<tr>
<td>221</td>
<td>-0.000267</td>
</tr>
<tr>
<td>222</td>
<td>0.000167</td>
</tr>
<tr>
<td>223</td>
<td>-0.000142</td>
</tr>
<tr>
<td>224</td>
<td>0.000111</td>
</tr>
<tr>
<td>225</td>
<td>0.000051</td>
</tr>
<tr>
<td>226</td>
<td>0.000084</td>
</tr>
<tr>
<td>227</td>
<td>-0.000194</td>
</tr>
<tr>
<td>228</td>
<td>0.000333</td>
</tr>
<tr>
<td>229</td>
<td>-0.000205</td>
</tr>
<tr>
<td>22A</td>
<td>0.000032</td>
</tr>
<tr>
<td>22B</td>
<td>-0.000026</td>
</tr>
<tr>
<td>22C</td>
<td>0.000065</td>
</tr>
<tr>
<td>22D</td>
<td>-0.000162</td>
</tr>
<tr>
<td>22E</td>
<td>0.000260</td>
</tr>
<tr>
<td>22F</td>
<td>-0.000362</td>
</tr>
<tr>
<td>230</td>
<td>0.000374</td>
</tr>
<tr>
<td>231</td>
<td>-0.000245</td>
</tr>
<tr>
<td>232</td>
<td>0.000281</td>
</tr>
<tr>
<td>233</td>
<td>-0.000276</td>
</tr>
<tr>
<td>234</td>
<td>0.000223</td>
</tr>
<tr>
<td>235</td>
<td>-0.000175</td>
</tr>
<tr>
<td>236</td>
<td>0.000079</td>
</tr>
<tr>
<td>237</td>
<td>0.000090</td>
</tr>
<tr>
<td>238</td>
<td>0.000100</td>
</tr>
<tr>
<td>239</td>
<td>-0.000481</td>
</tr>
<tr>
<td>23A</td>
<td>0.000359</td>
</tr>
<tr>
<td>23B</td>
<td>-0.000143</td>
</tr>
<tr>
<td>23C</td>
<td>0.000161</td>
</tr>
<tr>
<td>23D</td>
<td>-0.000188</td>
</tr>
<tr>
<td>23E</td>
<td>0.000104</td>
</tr>
<tr>
<td>23F</td>
<td>-0.000068</td>
</tr>
<tr>
<td>240</td>
<td>0.000166</td>
</tr>
<tr>
<td>241</td>
<td>-0.000170</td>
</tr>
<tr>
<td>242</td>
<td>0.000225</td>
</tr>
<tr>
<td>243</td>
<td>-0.000170</td>
</tr>
<tr>
<td>244</td>
<td>0.000133</td>
</tr>
<tr>
<td>245</td>
<td>-0.000295</td>
</tr>
<tr>
<td>246</td>
<td>0.000305</td>
</tr>
<tr>
<td>247</td>
<td>-0.000255</td>
</tr>
<tr>
<td>248</td>
<td>0.000137</td>
</tr>
<tr>
<td>249</td>
<td>-0.000053</td>
</tr>
<tr>
<td>24A</td>
<td>0.000090</td>
</tr>
<tr>
<td>24B</td>
<td>-0.000091</td>
</tr>
<tr>
<td>24C</td>
<td>0.000255</td>
</tr>
<tr>
<td>24D</td>
<td>-0.000233</td>
</tr>
<tr>
<td>24E</td>
<td>0.000108</td>
</tr>
<tr>
<td>24F</td>
<td>-0.000056</td>
</tr>
<tr>
<td>250</td>
<td>0.000273</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>251:</td>
<td>-0.000353</td>
</tr>
<tr>
<td>252:</td>
<td>0.000194</td>
</tr>
<tr>
<td>253:</td>
<td>-0.000192</td>
</tr>
<tr>
<td>254:</td>
<td>0.000350</td>
</tr>
<tr>
<td>255:</td>
<td>-0.000258</td>
</tr>
<tr>
<td>256:</td>
<td>0.000149</td>
</tr>
<tr>
<td>257:</td>
<td>-0.000100</td>
</tr>
<tr>
<td>258:</td>
<td>0.000050</td>
</tr>
<tr>
<td>259:</td>
<td>-0.000120</td>
</tr>
<tr>
<td>25A:</td>
<td>-0.000085</td>
</tr>
<tr>
<td>25B:</td>
<td>0.000087</td>
</tr>
<tr>
<td>25C:</td>
<td>0.000130</td>
</tr>
<tr>
<td>25D:</td>
<td>-0.000192</td>
</tr>
<tr>
<td>25E:</td>
<td>0.000002</td>
</tr>
<tr>
<td>25F:</td>
<td>-0.000069</td>
</tr>
<tr>
<td>260:</td>
<td>0.000113</td>
</tr>
<tr>
<td>261:</td>
<td>-0.000280</td>
</tr>
<tr>
<td>262:</td>
<td>0.000212</td>
</tr>
<tr>
<td>263:</td>
<td>-0.000129</td>
</tr>
<tr>
<td>264:</td>
<td>0.000107</td>
</tr>
<tr>
<td>265:</td>
<td>-0.000190</td>
</tr>
<tr>
<td>266:</td>
<td>0.000234</td>
</tr>
<tr>
<td>267:</td>
<td>-0.000203</td>
</tr>
<tr>
<td>268:</td>
<td>0.000128</td>
</tr>
<tr>
<td>269:</td>
<td>-0.000259</td>
</tr>
<tr>
<td>26A:</td>
<td>0.000106</td>
</tr>
<tr>
<td>26B:</td>
<td>-0.000048</td>
</tr>
<tr>
<td>26C:</td>
<td>-0.000024</td>
</tr>
<tr>
<td>26D:</td>
<td>-0.000016</td>
</tr>
<tr>
<td>26E:</td>
<td>0.000223</td>
</tr>
<tr>
<td>26F:</td>
<td>-0.000167</td>
</tr>
<tr>
<td>270:</td>
<td>0.000102</td>
</tr>
<tr>
<td>271:</td>
<td>-0.000196</td>
</tr>
<tr>
<td>272:</td>
<td>0.000077</td>
</tr>
<tr>
<td>273:</td>
<td>-0.000043</td>
</tr>
<tr>
<td>274:</td>
<td>0.000035</td>
</tr>
<tr>
<td>275:</td>
<td>-0.000272</td>
</tr>
<tr>
<td>276:</td>
<td>0.000174</td>
</tr>
<tr>
<td>277:</td>
<td>-0.000003</td>
</tr>
<tr>
<td>278:</td>
<td>-0.000126</td>
</tr>
<tr>
<td>279:</td>
<td>-0.000103</td>
</tr>
<tr>
<td>27A:</td>
<td>0.000109</td>
</tr>
<tr>
<td>27B:</td>
<td>-0.000063</td>
</tr>
<tr>
<td>27C:</td>
<td>0.000216</td>
</tr>
<tr>
<td>27D:</td>
<td>-0.000465</td>
</tr>
<tr>
<td>27E:</td>
<td>0.000379</td>
</tr>
<tr>
<td>27F:</td>
<td>-0.000262</td>
</tr>
<tr>
<td>280:</td>
<td>0.000635</td>
</tr>
<tr>
<td>281:</td>
<td>-0.000014</td>
</tr>
<tr>
<td>282:</td>
<td>0.000193</td>
</tr>
<tr>
<td>283:</td>
<td>-0.000470</td>
</tr>
<tr>
<td>284:</td>
<td>0.000474</td>
</tr>
<tr>
<td>285:</td>
<td>-0.000297</td>
</tr>
<tr>
<td>286:</td>
<td>0.000149</td>
</tr>
<tr>
<td>287:</td>
<td>-0.000166</td>
</tr>
<tr>
<td>288:</td>
<td>-0.000011</td>
</tr>
<tr>
<td>289:</td>
<td>-0.000087</td>
</tr>
<tr>
<td>28A:</td>
<td>0.000101</td>
</tr>
<tr>
<td>28B:</td>
<td>-0.000069</td>
</tr>
<tr>
<td>28C:</td>
<td>-0.000125</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001</td>
<td>-0.000032</td>
</tr>
<tr>
<td>000002</td>
<td>0.000077</td>
</tr>
<tr>
<td>000003</td>
<td>-0.000164</td>
</tr>
<tr>
<td>000004</td>
<td>0.000070</td>
</tr>
<tr>
<td>000005</td>
<td>-0.000136</td>
</tr>
<tr>
<td>000006</td>
<td>0.000062</td>
</tr>
<tr>
<td>000007</td>
<td>-0.000183</td>
</tr>
<tr>
<td>000008</td>
<td>0.000166</td>
</tr>
<tr>
<td>000009</td>
<td>-0.000273</td>
</tr>
<tr>
<td>00000A</td>
<td>0.000133</td>
</tr>
<tr>
<td>00000B</td>
<td>-0.000076</td>
</tr>
<tr>
<td>00000C</td>
<td>0.000045</td>
</tr>
<tr>
<td>00000D</td>
<td>-0.000234</td>
</tr>
<tr>
<td>00000E</td>
<td>0.000066</td>
</tr>
<tr>
<td>00000F</td>
<td>0.000072</td>
</tr>
<tr>
<td>000010</td>
<td>0.000123</td>
</tr>
<tr>
<td>000011</td>
<td>-0.000304</td>
</tr>
<tr>
<td>000012</td>
<td>0.000137</td>
</tr>
<tr>
<td>000013</td>
<td>-0.000206</td>
</tr>
<tr>
<td>000014</td>
<td>0.000196</td>
</tr>
<tr>
<td>000015</td>
<td>-0.000150</td>
</tr>
<tr>
<td>000016</td>
<td>0.000181</td>
</tr>
<tr>
<td>000017</td>
<td>-0.000160</td>
</tr>
<tr>
<td>000018</td>
<td>0.000106</td>
</tr>
<tr>
<td>000019</td>
<td>-0.000262</td>
</tr>
<tr>
<td>00001A</td>
<td>0.000167</td>
</tr>
<tr>
<td>00001B</td>
<td>-0.000173</td>
</tr>
<tr>
<td>00001C</td>
<td>0.000193</td>
</tr>
<tr>
<td>00001D</td>
<td>-0.000283</td>
</tr>
<tr>
<td>00001E</td>
<td>0.000117</td>
</tr>
<tr>
<td>00001F</td>
<td>0.000195</td>
</tr>
<tr>
<td>000020</td>
<td>0.000140</td>
</tr>
<tr>
<td>000021</td>
<td>-0.000215</td>
</tr>
<tr>
<td>000022</td>
<td>0.000044</td>
</tr>
<tr>
<td>000023</td>
<td>0.000047</td>
</tr>
<tr>
<td>000024</td>
<td>0.000028</td>
</tr>
<tr>
<td>000025</td>
<td>-0.000322</td>
</tr>
<tr>
<td>000026</td>
<td>0.000207</td>
</tr>
<tr>
<td>000027</td>
<td>-0.000082</td>
</tr>
<tr>
<td>000028</td>
<td>0.000125</td>
</tr>
<tr>
<td>000029</td>
<td>-0.000239</td>
</tr>
<tr>
<td>00002A</td>
<td>0.000078</td>
</tr>
<tr>
<td>00002B</td>
<td>0.000017</td>
</tr>
<tr>
<td>00002C</td>
<td>0.000128</td>
</tr>
<tr>
<td>00002D</td>
<td>-0.000179</td>
</tr>
<tr>
<td>00002E</td>
<td>0.000101</td>
</tr>
<tr>
<td>00002F</td>
<td>-0.000071</td>
</tr>
<tr>
<td>000030</td>
<td>0.000359</td>
</tr>
<tr>
<td>000031</td>
<td>-0.000240</td>
</tr>
<tr>
<td>000032</td>
<td>0.000054</td>
</tr>
<tr>
<td>000033</td>
<td>0.000079</td>
</tr>
<tr>
<td>000034</td>
<td>0.001402</td>
</tr>
<tr>
<td>000035</td>
<td>-0.000261</td>
</tr>
<tr>
<td>000036</td>
<td>0.000127</td>
</tr>
<tr>
<td>000037</td>
<td>-0.000113</td>
</tr>
<tr>
<td>000038</td>
<td>0.000268</td>
</tr>
<tr>
<td>000039</td>
<td>-0.000143</td>
</tr>
<tr>
<td>00003A</td>
<td>0.000007</td>
</tr>
<tr>
<td>00003B</td>
<td>0.000027</td>
</tr>
<tr>
<td>00003C</td>
<td>0.000243</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>0.000365</td>
</tr>
<tr>
<td>201</td>
<td>0.000336</td>
</tr>
<tr>
<td>202</td>
<td>0.000091</td>
</tr>
<tr>
<td>203</td>
<td>-0.000143</td>
</tr>
<tr>
<td>204</td>
<td>0.000220</td>
</tr>
<tr>
<td>205</td>
<td>-0.000286</td>
</tr>
<tr>
<td>206</td>
<td>0.000251</td>
</tr>
<tr>
<td>207</td>
<td>0.000001</td>
</tr>
<tr>
<td>208</td>
<td>-0.000069</td>
</tr>
<tr>
<td>209</td>
<td>-0.000222</td>
</tr>
<tr>
<td>20A</td>
<td>0.000336</td>
</tr>
<tr>
<td>20B</td>
<td>-0.000071</td>
</tr>
<tr>
<td>20C</td>
<td>0.000122</td>
</tr>
<tr>
<td>20D</td>
<td>-0.000540</td>
</tr>
<tr>
<td>20E</td>
<td>0.000304</td>
</tr>
<tr>
<td>20F</td>
<td>-0.000122</td>
</tr>
<tr>
<td>20G</td>
<td>-0.000134</td>
</tr>
<tr>
<td>20H</td>
<td>-0.000114</td>
</tr>
<tr>
<td>20I</td>
<td>0.000073</td>
</tr>
<tr>
<td>20J</td>
<td>-0.000040</td>
</tr>
<tr>
<td>20K</td>
<td>-0.000108</td>
</tr>
<tr>
<td>20L</td>
<td>0.000033</td>
</tr>
<tr>
<td>20M</td>
<td>0.000173</td>
</tr>
<tr>
<td>20N</td>
<td>0.000049</td>
</tr>
<tr>
<td>20O</td>
<td>0.000019</td>
</tr>
<tr>
<td>20P</td>
<td>-0.000234</td>
</tr>
<tr>
<td>20Q</td>
<td>0.000176</td>
</tr>
<tr>
<td>20R</td>
<td>-0.000281</td>
</tr>
<tr>
<td>20S</td>
<td>0.000200</td>
</tr>
<tr>
<td>20T</td>
<td>-0.000174</td>
</tr>
<tr>
<td>20U</td>
<td>0.000042</td>
</tr>
<tr>
<td>20V</td>
<td>-0.000086</td>
</tr>
<tr>
<td>20W</td>
<td>0.000063</td>
</tr>
<tr>
<td>20X</td>
<td>-0.000160</td>
</tr>
<tr>
<td>20Y</td>
<td>0.000127</td>
</tr>
<tr>
<td>20Z</td>
<td>-0.000134</td>
</tr>
<tr>
<td>20AA</td>
<td>0.000058</td>
</tr>
<tr>
<td>20AB</td>
<td>-0.000211</td>
</tr>
<tr>
<td>20AC</td>
<td>0.000104</td>
</tr>
<tr>
<td>20AD</td>
<td>-0.000031</td>
</tr>
<tr>
<td>20AE</td>
<td>-0.000134</td>
</tr>
<tr>
<td>20AF</td>
<td>-0.000003</td>
</tr>
<tr>
<td>20B</td>
<td>-0.000005</td>
</tr>
<tr>
<td>20B</td>
<td>-0.000026</td>
</tr>
<tr>
<td>20C</td>
<td>0.000093</td>
</tr>
<tr>
<td>20D</td>
<td>-0.000015</td>
</tr>
<tr>
<td>20E</td>
<td>-0.000090</td>
</tr>
<tr>
<td>20F</td>
<td>0.000570</td>
</tr>
<tr>
<td>20G</td>
<td>0.000537</td>
</tr>
<tr>
<td>20H</td>
<td>-0.000111</td>
</tr>
<tr>
<td>20I</td>
<td>-0.000307</td>
</tr>
<tr>
<td>20J</td>
<td>0.000133</td>
</tr>
<tr>
<td>20K</td>
<td>-0.000116</td>
</tr>
<tr>
<td>20L</td>
<td>0.000032</td>
</tr>
</tbody>
</table>

DNL Error, SN = 4130 (Continued)

6-273
| 305: | -0.000166: | 306: | 0.000148: | 307: | -0.000214: | 308: | 0.000217: | 309: | -0.000323: | 30A: | 0.000125: | 30B: | -0.000224: | 30C: | 0.000131: | 30D: | -0.000248: | 30E: | 0.000148: | 310: | -0.000333: | 311: | 0.000249: | 312: | -0.000134: | 313: | 0.000035: | 314: | -0.000244: | 315: | 0.000141: | 316: | -0.000051: | 317: | 0.000094: | 318: | -0.000189: | 319: | 0.000070: | 31A: | 0.000012: | 31B: | 0.000158: | 31C: | -0.000233: | 31D: | 0.000096: | 31E: | -0.000286: | 31F: | 0.000173: | 320: | -0.000229: | 321: | 0.000137: | 322: | -0.000112: | 323: | 0.000086: | 324: | -0.000205: | 325: | 0.000183: | 326: | -0.000116: | 327: | 0.000054: | 328: | -0.000096: | 329: | 0.000149: | 32A: | -0.000114: | 32B: | 0.000118: | 32C: | -0.000156: | 32D: | 0.000086: | 32E: | -0.000081: | 32F: | 0.000352: | 330: | 0.000245: | 331: | 0.000125: | 332: | -0.000064: | 333: | 0.000270: | 334: | -0.000240: | 335: | 0.000116: | 336: | -0.000001: | 337: | 0.000027: | 338: | -0.000190: | 339: | 0.000010: | 33A: | -0.000001: | 33B: | 0.000048: | 33C: | -0.000128: | 33D: | -0.000332: | 33E: | 0.000222: | 33F: | -0.000265: | 340: | 0.001727: |

DNL Error, SN = 4130 (Continued)
DNL Error, SN = 4130 (Continued)
| 37B: | -0.000158:   |   |
| 37E: | -0.000127:   |   |
| 37F: | 0.000509:    |   |
| 380: | -0.000187:   |   |
| 381: | -0.000301:   |   |
| 382: | 0.000244:    |   |
| 383: | -0.000066:   |   |
| 384: | 0.000016:    |   |
| 385: | -0.000098:   |   |
| 386: | 0.000121:    |   |
| 387: | -0.000090:   |   |
| 388: | -0.000072:   |   |
| 389: | -0.000026:   |   |
| 38A: | 0.000078:    |   |
| 38B: | -0.000107:   |   |
| 38C: | -0.000009:   |   |
| 38D: | -0.000282:   |   |
| 38E: | 0.000292:    |   |
| 38F: | -0.000294:   |   |
| 390: | 0.000298:    |   |
| 391: | -0.000572:   |   |
| 392: | 0.000375:    |   |
| 393: | -0.000192:   |   |
| 394: | 0.000255:    |   |
| 395: | -0.000245:   |   |
| 396: | -0.000019:   |   |
| 397: | 0.000076:    |   |
| 398: | 0.000104:    |   |
| 399: | -0.000252:   |   |
| 39A: | 0.000168:    |   |
| 39B: | -0.000075:   |   |
| 39C: | 0.000243:    |   |
| 39D: | -0.000288:   |   |
| 39E: | 0.000147:    |   |
| 39F: | -0.000093:   |   |
| 3A0: | 0.000151:    |   |
| 3A1: | -0.000219:   |   |
| 3A2: | 0.000074:    |   |
| 3A3: | -0.000066:   |   |
| 3A4: | 0.000245:    |   |
| 3A5: | -0.000101:   |   |
| 3A6: | -0.000114:   |   |
| 3A7: | 0.000038:    |   |
| 3A8: | 0.000047:    |   |
| 3A9: | -0.000286:   |   |
| 3AA: | 0.000150:    |   |
| 3AB: | -0.000039:   |   |
| 3AC: | 0.000052:    |   |
| 3AD: | -0.000079:   |   |
| 3AE: | 0.000003:    |   |
| 3AF: | 0.000036:    |   |
| 3B0: | -0.000067:   |   |
| 3B1: | -0.000262:   |   |
| 3B2: | 0.000178:    |   |
| 3B3: | -0.000095:   |   |
| 3B4: | 0.000242:    |   |
| 3B5: | -0.000344:   |   |
| 3B6: | 0.000216:    |   |
| 3B7: | 0.000004:    |   |
| 3B8: | -0.000152:   |   |

DNL Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>0.000085</td>
</tr>
<tr>
<td>301</td>
<td>0.000078</td>
</tr>
<tr>
<td>302</td>
<td>0.000103</td>
</tr>
<tr>
<td>303</td>
<td>0.000155</td>
</tr>
<tr>
<td>304</td>
<td>0.000028</td>
</tr>
<tr>
<td>305</td>
<td>0.000093</td>
</tr>
<tr>
<td>306</td>
<td>0.000145</td>
</tr>
<tr>
<td>307</td>
<td>0.000069</td>
</tr>
<tr>
<td>308</td>
<td>0.000083</td>
</tr>
<tr>
<td>309</td>
<td>0.000097</td>
</tr>
<tr>
<td>310</td>
<td>0.000162</td>
</tr>
<tr>
<td>311</td>
<td>0.000098</td>
</tr>
<tr>
<td>312</td>
<td>0.000208</td>
</tr>
<tr>
<td>313</td>
<td>0.000096</td>
</tr>
<tr>
<td>314</td>
<td>0.000049</td>
</tr>
<tr>
<td>315</td>
<td>0.000018</td>
</tr>
<tr>
<td>316</td>
<td>0.000091</td>
</tr>
<tr>
<td>317</td>
<td>0.000025</td>
</tr>
<tr>
<td>318</td>
<td>0.000031</td>
</tr>
<tr>
<td>319</td>
<td>0.000054</td>
</tr>
<tr>
<td>320</td>
<td>0.000067</td>
</tr>
<tr>
<td>321</td>
<td>0.000196</td>
</tr>
<tr>
<td>322</td>
<td>0.000126</td>
</tr>
<tr>
<td>323</td>
<td>0.000002</td>
</tr>
<tr>
<td>324</td>
<td>0.000031</td>
</tr>
<tr>
<td>325</td>
<td>0.000276</td>
</tr>
<tr>
<td>326</td>
<td>0.000056</td>
</tr>
<tr>
<td>327</td>
<td>0.000007</td>
</tr>
<tr>
<td>328</td>
<td>0.000073</td>
</tr>
<tr>
<td>329</td>
<td>0.000073</td>
</tr>
<tr>
<td>330</td>
<td>0.000118</td>
</tr>
<tr>
<td>331</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

**DNL Error, SN = 4130 (Continued)**
Repeatability Error, SN = 4130

<table>
<thead>
<tr>
<th>y_{min}</th>
<th>-0.0020</th>
<th>+ 0 -</th>
<th>y_{max}</th>
<th>0.0020</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1A: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1B: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1C: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1E: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29: 0.001375:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2A: 0.001375:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2B: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2C: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2E: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2F: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30: 0.001175:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32: 0.001075:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33: 0.000975:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34: 0.001275:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex</td>
<td>Value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3B</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>0.001115</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3E</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3F</td>
<td>0.001375</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4A</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4B</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>0.001375</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4D</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4E</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4F</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>0.000975</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>0.000975</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5A</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5B</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5D</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5E</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5F</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>0.000975</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.001275</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>0.001375</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6A</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6B</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6C</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6D</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6E</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6F</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>0.001075</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Repeatability Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>SN</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>0.001375</td>
</tr>
<tr>
<td>72</td>
<td>0.001175</td>
</tr>
<tr>
<td>73</td>
<td>0.001175</td>
</tr>
<tr>
<td>74</td>
<td>0.001175</td>
</tr>
<tr>
<td>75</td>
<td>0.001175</td>
</tr>
<tr>
<td>76</td>
<td>0.001175</td>
</tr>
<tr>
<td>77</td>
<td>0.001275</td>
</tr>
<tr>
<td>78</td>
<td>0.001175</td>
</tr>
<tr>
<td>79</td>
<td>0.001275</td>
</tr>
<tr>
<td>7A</td>
<td>0.001375</td>
</tr>
<tr>
<td>7B</td>
<td>0.001375</td>
</tr>
<tr>
<td>7C</td>
<td>0.001375</td>
</tr>
<tr>
<td>7D</td>
<td>0.001375</td>
</tr>
<tr>
<td>7E</td>
<td>0.001175</td>
</tr>
<tr>
<td>7F</td>
<td>0.001175</td>
</tr>
<tr>
<td>80</td>
<td>0.001275</td>
</tr>
<tr>
<td>81</td>
<td>0.001175</td>
</tr>
<tr>
<td>82</td>
<td>0.001275</td>
</tr>
<tr>
<td>83</td>
<td>0.001275</td>
</tr>
<tr>
<td>84</td>
<td>0.001275</td>
</tr>
<tr>
<td>85</td>
<td>0.001175</td>
</tr>
<tr>
<td>86</td>
<td>0.001175</td>
</tr>
<tr>
<td>87</td>
<td>0.001275</td>
</tr>
<tr>
<td>88</td>
<td>0.001175</td>
</tr>
<tr>
<td>89</td>
<td>0.001175</td>
</tr>
<tr>
<td>8A</td>
<td>0.001175</td>
</tr>
<tr>
<td>8B</td>
<td>0.001175</td>
</tr>
<tr>
<td>8C</td>
<td>0.001175</td>
</tr>
<tr>
<td>8D</td>
<td>0.001175</td>
</tr>
<tr>
<td>8E</td>
<td>0.001275</td>
</tr>
<tr>
<td>8F</td>
<td>0.001175</td>
</tr>
<tr>
<td>90</td>
<td>0.001175</td>
</tr>
<tr>
<td>91</td>
<td>0.001175</td>
</tr>
<tr>
<td>92</td>
<td>0.001275</td>
</tr>
<tr>
<td>93</td>
<td>0.001175</td>
</tr>
<tr>
<td>94</td>
<td>0.001175</td>
</tr>
<tr>
<td>95</td>
<td>0.001175</td>
</tr>
<tr>
<td>96</td>
<td>0.001175</td>
</tr>
<tr>
<td>97</td>
<td>0.001175</td>
</tr>
<tr>
<td>98</td>
<td>0.001175</td>
</tr>
<tr>
<td>99</td>
<td>0.001175</td>
</tr>
<tr>
<td>9A</td>
<td>0.001275</td>
</tr>
<tr>
<td>9B</td>
<td>0.001175</td>
</tr>
<tr>
<td>9C</td>
<td>0.001175</td>
</tr>
<tr>
<td>9D</td>
<td>0.001275</td>
</tr>
<tr>
<td>9E</td>
<td>0.001275</td>
</tr>
<tr>
<td>9F</td>
<td>0.001175</td>
</tr>
<tr>
<td>A0</td>
<td>0.001175</td>
</tr>
<tr>
<td>A1</td>
<td>0.001175</td>
</tr>
<tr>
<td>A2</td>
<td>0.001175</td>
</tr>
<tr>
<td>A3</td>
<td>0.001275</td>
</tr>
<tr>
<td>A4</td>
<td>0.001175</td>
</tr>
<tr>
<td>A5</td>
<td>0.001175</td>
</tr>
<tr>
<td>A6</td>
<td>0.001275</td>
</tr>
<tr>
<td>A7</td>
<td>0.001175</td>
</tr>
<tr>
<td>A8</td>
<td>0.001175</td>
</tr>
<tr>
<td>A9</td>
<td>0.001175</td>
</tr>
<tr>
<td>AA</td>
<td>0.001175</td>
</tr>
<tr>
<td>AB</td>
<td>0.001175</td>
</tr>
<tr>
<td>AC</td>
<td>0.001175</td>
</tr>
</tbody>
</table>

Repeatability Error, SN = 4130 (Continued)
| AD: 0.001075 | | | |
| AE: 0.001175 | | | |
| AF: 0.001075 | | | |
| AO: 0.001075 | | | |
| AI: 0.001175 | | | |
| A2: 0.001275 | | | |
| A3: 0.001275 | | | |
| A4: 0.000975 | | | |
| A5: 0.001175 | | | |
| A6: 0.001075 | | | |
| A7: 0.001175 | | | |
| A8: 0.001275 | | | |
| A9: 0.001275 | | | |
| BA: 0.001175 | | | |
| BB: 0.001175 | | | |
| BC: 0.001075 | | | |
| BD: 0.001175 | | | |
| BE: 0.001175 | | | |
| BF: 0.001175 | | | |
| C0: 0.001275 | | | |
| C1: 0.001275 | | | |
| C2: 0.001075 | | | |
| C3: 0.000975 | | | |
| C4: 0.001175 | | | |
| C5: 0.001175 | | | |
| C6: 0.001175 | | | |
| C7: 0.001275 | | | |
| C8: 0.001175 | | | |
| C9: 0.001175 | | | |
| CA: 0.001075 | | | |
| CB: 0.001375 | | | |
| CC: 0.001275 | | | |
| CD: 0.001275 | | | |
| CE: 0.001175 | | | |
| CF: 0.001275 | | | |
| D0: 0.001175 | | | |
| D1: 0.001075 | | | |
| D2: 0.001275 | | | |
| D3: 0.001275 | | | |
| D4: 0.001175 | | | |
| D5: 0.001175 | | | |
| D6: 0.001075 | | | |
| D7: 0.001275 | | | |
| D8: 0.001175 | | | |
| D9: 0.001275 | | | |
| DA: 0.001175 | | | |
| DB: 0.001175 | | | |
| DC: 0.001275 | | | |
| DD: 0.001275 | | | |
| DE: 0.001275 | | | |
| DF: 0.001275 | | | |
| DG: 0.001175 | | | |
| EH: 0.000975 | | | |
| E1: 0.001275 | | | |
| E2: 0.001275 | | | |
| E3: 0.001175 | | | |
| E4: 0.001175 | | | |
| E5: 0.001075 | | | |
| E6: 0.001175 | | | |
| E7: 0.001175 | | | |
| EB: 0.001175 | | | |

Repeatability Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Location</th>
<th>Repeatability Error</th>
<th>SN</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>0.001375</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>0.001075</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>0.001275</td>
<td></td>
</tr>
<tr>
<td>A7</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td>0.001375</td>
<td></td>
</tr>
<tr>
<td>A11</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A12</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A13</td>
<td>0.001275</td>
<td></td>
</tr>
<tr>
<td>A14</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A15</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A16</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A17</td>
<td>0.001375</td>
<td></td>
</tr>
<tr>
<td>A18</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A19</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A20</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A21</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A22</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A23</td>
<td>0.001175</td>
<td></td>
</tr>
<tr>
<td>A24</td>
<td>0.001175</td>
<td></td>
</tr>
</tbody>
</table>

Repeatability Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Repeatability Error, SN = 4130 (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>125: 0.001175:</td>
</tr>
<tr>
<td>126: 0.001175:</td>
</tr>
<tr>
<td>127: 0.001275:</td>
</tr>
<tr>
<td>128: 0.001075:</td>
</tr>
<tr>
<td>129: 0.001175:</td>
</tr>
<tr>
<td>12A: 0.001275:</td>
</tr>
<tr>
<td>12B: 0.001175:</td>
</tr>
<tr>
<td>12C: 0.001175:</td>
</tr>
<tr>
<td>12D: 0.001275:</td>
</tr>
<tr>
<td>12E: 0.001075:</td>
</tr>
<tr>
<td>12F: 0.001275:</td>
</tr>
<tr>
<td>130: 0.001175:</td>
</tr>
<tr>
<td>131: 0.001175:</td>
</tr>
<tr>
<td>132: 0.001075:</td>
</tr>
<tr>
<td>133: 0.001075:</td>
</tr>
<tr>
<td>134: 0.001275:</td>
</tr>
<tr>
<td>135: 0.001275:</td>
</tr>
<tr>
<td>136: 0.001275:</td>
</tr>
<tr>
<td>137: 0.001075:</td>
</tr>
<tr>
<td>138: 0.001175:</td>
</tr>
<tr>
<td>139: 0.001275:</td>
</tr>
<tr>
<td>13A: 0.001175:</td>
</tr>
<tr>
<td>13B: 0.001275:</td>
</tr>
<tr>
<td>13C: 0.001175:</td>
</tr>
<tr>
<td>13D: 0.001175:</td>
</tr>
<tr>
<td>13E: 0.001275:</td>
</tr>
<tr>
<td>13F: 0.001075:</td>
</tr>
<tr>
<td>140: 0.001075:</td>
</tr>
<tr>
<td>141: 0.001075:</td>
</tr>
<tr>
<td>142: 0.001075:</td>
</tr>
<tr>
<td>143: 0.001075:</td>
</tr>
<tr>
<td>144: 0.001175:</td>
</tr>
<tr>
<td>145: 0.001275:</td>
</tr>
<tr>
<td>146: 0.001375:</td>
</tr>
<tr>
<td>147: 0.001375:</td>
</tr>
<tr>
<td>148: 0.001075:</td>
</tr>
<tr>
<td>149: 0.001275:</td>
</tr>
<tr>
<td>14A: 0.001275:</td>
</tr>
<tr>
<td>14B: 0.001275:</td>
</tr>
<tr>
<td>14C: 0.001375:</td>
</tr>
<tr>
<td>14D: 0.001375:</td>
</tr>
<tr>
<td>14E: 0.001175:</td>
</tr>
<tr>
<td>14F: 0.001175:</td>
</tr>
<tr>
<td>150: 0.001175:</td>
</tr>
<tr>
<td>151: 0.001275:</td>
</tr>
<tr>
<td>152: 0.001375:</td>
</tr>
<tr>
<td>153: 0.001275:</td>
</tr>
<tr>
<td>154: 0.001275:</td>
</tr>
<tr>
<td>155: 0.001175:</td>
</tr>
<tr>
<td>156: 0.001175:</td>
</tr>
<tr>
<td>157: 0.001275:</td>
</tr>
<tr>
<td>158: 0.001175:</td>
</tr>
<tr>
<td>159: 0.001175:</td>
</tr>
<tr>
<td>15A: 0.001175:</td>
</tr>
<tr>
<td>15B: 0.001275:</td>
</tr>
<tr>
<td>15C: 0.001175:</td>
</tr>
<tr>
<td>15D: 0.001375:</td>
</tr>
<tr>
<td>15E: 0.001275:</td>
</tr>
<tr>
<td>15F: 0.001375:</td>
</tr>
<tr>
<td>160: 0.001375:</td>
</tr>
<tr>
<td>SN</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>161</td>
</tr>
<tr>
<td>162</td>
</tr>
<tr>
<td>163</td>
</tr>
<tr>
<td>164</td>
</tr>
<tr>
<td>165</td>
</tr>
<tr>
<td>166</td>
</tr>
<tr>
<td>167</td>
</tr>
<tr>
<td>168</td>
</tr>
<tr>
<td>169</td>
</tr>
<tr>
<td>170</td>
</tr>
<tr>
<td>171</td>
</tr>
<tr>
<td>172</td>
</tr>
<tr>
<td>173</td>
</tr>
<tr>
<td>174</td>
</tr>
<tr>
<td>175</td>
</tr>
<tr>
<td>176</td>
</tr>
<tr>
<td>177</td>
</tr>
<tr>
<td>178</td>
</tr>
<tr>
<td>179</td>
</tr>
<tr>
<td>180</td>
</tr>
<tr>
<td>181</td>
</tr>
<tr>
<td>182</td>
</tr>
<tr>
<td>183</td>
</tr>
<tr>
<td>184</td>
</tr>
<tr>
<td>185</td>
</tr>
<tr>
<td>186</td>
</tr>
<tr>
<td>187</td>
</tr>
<tr>
<td>188</td>
</tr>
<tr>
<td>189</td>
</tr>
<tr>
<td>190</td>
</tr>
<tr>
<td>191</td>
</tr>
<tr>
<td>192</td>
</tr>
<tr>
<td>193</td>
</tr>
<tr>
<td>194</td>
</tr>
<tr>
<td>195</td>
</tr>
<tr>
<td>196</td>
</tr>
<tr>
<td>197</td>
</tr>
<tr>
<td>198</td>
</tr>
<tr>
<td>199</td>
</tr>
<tr>
<td>19A</td>
</tr>
<tr>
<td>19B</td>
</tr>
<tr>
<td>19C</td>
</tr>
</tbody>
</table>

Repeatability Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Repeatability Error, SN = 4130 (Continued)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>190: 0.001275</td>
<td></td>
</tr>
<tr>
<td>192: 0.001275</td>
<td></td>
</tr>
<tr>
<td>19F: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1A0: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1A1: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1A2: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1A3: 0.001475</td>
<td></td>
</tr>
<tr>
<td>1A4: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1A5: 0.001075</td>
<td></td>
</tr>
<tr>
<td>1A6: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1A7: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1A8: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1A9: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1AA: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1AB: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1AC: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1AD: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1AE: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1AF: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1B0: 0.001075</td>
<td></td>
</tr>
<tr>
<td>1B1: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1B2: 0.000975</td>
<td></td>
</tr>
<tr>
<td>1B3: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1B4: 0.001075</td>
<td></td>
</tr>
<tr>
<td>1B5: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1B6: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1B7: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1B8: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1B9: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1BA: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1BB: 0.001475</td>
<td></td>
</tr>
<tr>
<td>1BC: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1BD: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1BE: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1BF: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1C0: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1C1: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1C2: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1C3: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1C4: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1C5: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1C6: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1C7: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1C8: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1C9: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1CA: 0.001075</td>
<td></td>
</tr>
<tr>
<td>1CB: 0.001075</td>
<td></td>
</tr>
<tr>
<td>1CC: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1CD: 0.001375</td>
<td></td>
</tr>
<tr>
<td>1CE: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1CF: 0.001075</td>
<td></td>
</tr>
<tr>
<td>1D0: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1D1: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1D2: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1D3: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1D4: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1D5: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1D6: 0.001275</td>
<td></td>
</tr>
<tr>
<td>1D7: 0.001175</td>
<td></td>
</tr>
<tr>
<td>1D8: 0.001275</td>
<td></td>
</tr>
</tbody>
</table>

270365–58
|    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    | Value |    |
|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|-------|----|
| 109 | 0.001275 | 110 | 0.001275 | 111 | 0.001175 |
| 112 | 0.001175 | 113 | 0.001275 | 114 | 0.001175 |
| 115 | 0.001275 | 116 | 0.001175 | 117 | 0.001275 |
| 118 | 0.001175 | 119 | 0.001175 | 120 | 0.001175 |
| 121 | 0.001175 | 122 | 0.001275 | 123 | 0.001175 |
| 124 | 0.001175 | 125 | 0.001175 | 126 | 0.001275 |
| 127 | 0.001275 | 128 | 0.001275 | 129 | 0.001175 |
| 130 | 0.001175 | 131 | 0.001175 | 132 | 0.001275 |
| 133 | 0.001275 | 134 | 0.001275 | 135 | 0.001275 |
| 136 | 0.001375 | 137 | 0.001375 | 138 | 0.001375 |
| 139 | 0.001375 | 140 | 0.001375 | 141 | 0.001375 |
| 142 | 0.001375 | 143 | 0.001375 | 144 | 0.001375 |
| 145 | 0.001375 | 146 | 0.001375 | 147 | 0.001375 |
| 148 | 0.001375 | 149 | 0.001375 | 150 | 0.001375 |
| 151 | 0.001375 | 152 | 0.001375 | 153 | 0.001375 |
| 154 | 0.001375 | 155 | 0.001375 | 156 | 0.001375 |
| 157 | 0.001375 | 158 | 0.001375 | 159 | 0.001375 |
| 160 | 0.001375 | 161 | 0.001375 | 162 | 0.001375 |
| 163 | 0.001375 | 164 | 0.001375 | 165 | 0.001375 |
| 166 | 0.001375 | 167 | 0.001375 | 168 | 0.001375 |
| 169 | 0.001375 | 170 | 0.001375 | 171 | 0.001375 |
| 172 | 0.001375 | 173 | 0.001375 | 174 | 0.001375 |
| 175 | 0.001375 | 176 | 0.001375 | 177 | 0.001375 |
| 178 | 0.001375 | 179 | 0.001375 | 180 | 0.001375 |
| 181 | 0.001375 | 182 | 0.001375 | 183 | 0.001375 |
| 184 | 0.001375 | 185 | 0.001375 | 186 | 0.001375 |
| 187 | 0.001375 | 188 | 0.001375 | 189 | 0.001375 |
| 190 | 0.001375 | 191 | 0.001375 | 192 | 0.001375 |
| 193 | 0.001375 | 194 | 0.001375 | 195 | 0.001375 |
| 196 | 0.001375 | 197 | 0.001375 | 198 | 0.001375 |
| 199 | 0.001375 | 200 | 0.001375 | 201 | 0.001375 |
| 202 | 0.001375 | 203 | 0.001375 | 204 | 0.001375 |
| 205 | 0.001375 | 206 | 0.001375 | 207 | 0.001375 |
| 208 | 0.001375 | 209 | 0.001375 | 210 | 0.001375 |
| 211 | 0.001375 | 212 | 0.001375 | 213 | 0.001375 |
| 214 | 0.001375 | 215 | 0.001375 | 216 | 0.001375 |

Repeatability Error, SN = 4130 (Continued)

6-286
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>215</td>
<td>0.001275</td>
</tr>
<tr>
<td>21b</td>
<td>0.001275</td>
</tr>
<tr>
<td>217</td>
<td>0.001275</td>
</tr>
<tr>
<td>218</td>
<td>0.001275</td>
</tr>
<tr>
<td>219</td>
<td>0.001375</td>
</tr>
<tr>
<td>21a</td>
<td>0.001175</td>
</tr>
<tr>
<td>21b</td>
<td>0.001275</td>
</tr>
<tr>
<td>21c</td>
<td>0.001275</td>
</tr>
<tr>
<td>21d</td>
<td>0.001275</td>
</tr>
<tr>
<td>21e</td>
<td>0.001175</td>
</tr>
<tr>
<td>21f</td>
<td>0.001175</td>
</tr>
<tr>
<td>220</td>
<td>0.001375</td>
</tr>
<tr>
<td>221</td>
<td>0.001275</td>
</tr>
<tr>
<td>222</td>
<td>0.001375</td>
</tr>
<tr>
<td>223</td>
<td>0.001375</td>
</tr>
<tr>
<td>224</td>
<td>0.001175</td>
</tr>
<tr>
<td>225</td>
<td>0.001375</td>
</tr>
<tr>
<td>226</td>
<td>0.001175</td>
</tr>
<tr>
<td>227</td>
<td>0.001375</td>
</tr>
<tr>
<td>228</td>
<td>0.001175</td>
</tr>
<tr>
<td>229</td>
<td>0.001275</td>
</tr>
<tr>
<td>22a</td>
<td>0.001175</td>
</tr>
<tr>
<td>22b</td>
<td>0.001275</td>
</tr>
<tr>
<td>22c</td>
<td>0.001275</td>
</tr>
<tr>
<td>22d</td>
<td>0.001175</td>
</tr>
<tr>
<td>22e</td>
<td>0.001175</td>
</tr>
<tr>
<td>22f</td>
<td>0.001075</td>
</tr>
<tr>
<td>230</td>
<td>0.001275</td>
</tr>
<tr>
<td>231</td>
<td>0.001175</td>
</tr>
<tr>
<td>232</td>
<td>0.001175</td>
</tr>
<tr>
<td>233</td>
<td>0.001275</td>
</tr>
<tr>
<td>234</td>
<td>0.001275</td>
</tr>
<tr>
<td>235</td>
<td>0.001375</td>
</tr>
<tr>
<td>236</td>
<td>0.001375</td>
</tr>
<tr>
<td>237</td>
<td>0.001275</td>
</tr>
<tr>
<td>238</td>
<td>0.001275</td>
</tr>
<tr>
<td>239</td>
<td>0.001175</td>
</tr>
<tr>
<td>23a</td>
<td>0.001175</td>
</tr>
<tr>
<td>23b</td>
<td>0.001175</td>
</tr>
<tr>
<td>23c</td>
<td>0.001175</td>
</tr>
<tr>
<td>23d</td>
<td>0.001375</td>
</tr>
<tr>
<td>23e</td>
<td>0.001175</td>
</tr>
<tr>
<td>23f</td>
<td>0.001275</td>
</tr>
<tr>
<td>240</td>
<td>0.001275</td>
</tr>
<tr>
<td>241</td>
<td>0.001275</td>
</tr>
<tr>
<td>242</td>
<td>0.001275</td>
</tr>
<tr>
<td>243</td>
<td>0.001275</td>
</tr>
<tr>
<td>244</td>
<td>0.001275</td>
</tr>
<tr>
<td>245</td>
<td>0.001375</td>
</tr>
<tr>
<td>246</td>
<td>0.001075</td>
</tr>
<tr>
<td>247</td>
<td>0.001175</td>
</tr>
<tr>
<td>248</td>
<td>0.001175</td>
</tr>
<tr>
<td>249</td>
<td>0.001375</td>
</tr>
<tr>
<td>24a</td>
<td>0.001175</td>
</tr>
<tr>
<td>24b</td>
<td>0.001275</td>
</tr>
<tr>
<td>24c</td>
<td>0.001075</td>
</tr>
<tr>
<td>24d</td>
<td>0.001175</td>
</tr>
<tr>
<td>24e</td>
<td>0.001375</td>
</tr>
<tr>
<td>24f</td>
<td>0.001375</td>
</tr>
<tr>
<td>250</td>
<td>0.001275</td>
</tr>
</tbody>
</table>

Repeatability Error, SN = 4130 (Continued)
Repeatability Error, SN = 4130 (Continued)
Repeatability Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>200:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>201:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>204:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>266:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>267:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>268:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>269:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>270:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>271:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>272:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>273:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>274:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>275:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>276:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>277:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>278:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>279:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>280:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>281:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>282:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>283:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>284:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>285:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>286:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>287:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>288:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>289:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>290:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>291:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>292:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>293:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>294:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>295:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>296:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>297:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>298:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>299:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>301:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>302:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>303:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>304:</td>
<td>0.001175</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Repeatability Error, SN = 4130 (Continued)
<table>
<thead>
<tr>
<th>SN</th>
<th>Repeatability Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>305</td>
<td>0.001475</td>
</tr>
<tr>
<td>306</td>
<td>0.001275</td>
</tr>
<tr>
<td>307</td>
<td>0.001375</td>
</tr>
<tr>
<td>308</td>
<td>0.001375</td>
</tr>
<tr>
<td>309</td>
<td>0.001275</td>
</tr>
<tr>
<td>310</td>
<td>0.001175</td>
</tr>
<tr>
<td>311</td>
<td>0.001175</td>
</tr>
<tr>
<td>312</td>
<td>0.001275</td>
</tr>
<tr>
<td>313</td>
<td>0.001275</td>
</tr>
<tr>
<td>314</td>
<td>0.001375</td>
</tr>
<tr>
<td>315</td>
<td>0.001275</td>
</tr>
<tr>
<td>316</td>
<td>0.001275</td>
</tr>
<tr>
<td>317</td>
<td>0.001175</td>
</tr>
<tr>
<td>318</td>
<td>0.001175</td>
</tr>
<tr>
<td>319</td>
<td>0.001375</td>
</tr>
<tr>
<td>320</td>
<td>0.001375</td>
</tr>
<tr>
<td>321</td>
<td>0.001375</td>
</tr>
<tr>
<td>322</td>
<td>0.001375</td>
</tr>
<tr>
<td>323</td>
<td>0.001275</td>
</tr>
<tr>
<td>324</td>
<td>0.001174</td>
</tr>
<tr>
<td>325</td>
<td>0.001575</td>
</tr>
<tr>
<td>326</td>
<td>0.001275</td>
</tr>
<tr>
<td>327</td>
<td>0.001475</td>
</tr>
<tr>
<td>328</td>
<td>0.001174</td>
</tr>
<tr>
<td>329</td>
<td>0.001375</td>
</tr>
<tr>
<td>330</td>
<td>0.001375</td>
</tr>
<tr>
<td>331</td>
<td>0.001475</td>
</tr>
<tr>
<td>332</td>
<td>0.001275</td>
</tr>
<tr>
<td>333</td>
<td>0.001275</td>
</tr>
<tr>
<td>334</td>
<td>0.001575</td>
</tr>
<tr>
<td>335</td>
<td>0.001475</td>
</tr>
<tr>
<td>336</td>
<td>0.001174</td>
</tr>
<tr>
<td>337</td>
<td>0.001174</td>
</tr>
<tr>
<td>338</td>
<td>0.001275</td>
</tr>
<tr>
<td>339</td>
<td>0.001275</td>
</tr>
<tr>
<td>340</td>
<td>0.001275</td>
</tr>
<tr>
<td>341</td>
<td>0.001275</td>
</tr>
<tr>
<td>342</td>
<td>0.001275</td>
</tr>
<tr>
<td>343</td>
<td>0.001275</td>
</tr>
<tr>
<td>344</td>
<td>0.001275</td>
</tr>
<tr>
<td>345</td>
<td>0.001275</td>
</tr>
<tr>
<td>346</td>
<td>0.001275</td>
</tr>
<tr>
<td>347</td>
<td>0.001275</td>
</tr>
<tr>
<td>348</td>
<td>0.001275</td>
</tr>
</tbody>
</table>

Repeatability Error, SN = 4130 (Continued)

6-291
Repeatability Error, SN = 4130 (Continued)
| 370: 0.001375 |   |   |
| 372: 0.001375 |   |   |
| 374: 0.001375 |   |   |
| 380: 0.001475 |   |   |
| 381: 0.001475 |   |   |
| 382: 0.001275 |   |   |
| 383: 0.001475 |   |   |
| 384: 0.001375 |   |   |
| 385: 0.001475 |   |   |
| 386: 0.001275 |   |   |
| 387: 0.001275 |   |   |
| 388: 0.001375 |   |   |
| 389: 0.001315 |   |   |
| 390: 0.001375 |   |   |
| 391: 0.001174 |   |   |
| 392: 0.001375 |   |   |
| 393: 0.001375 |   |   |
| 394: 0.001375 |   |   |
| 395: 0.001375 |   |   |
| 396: 0.001074 |   |   |
| 397: 0.001275 |   |   |
| 398: 0.001375 |   |   |
| 399: 0.001375 |   |   |
| 3A0: 0.001275 |   |   |
| 3A1: 0.001275 |   |   |
| 3A2: 0.001275 |   |   |
| 3A3: 0.001275 |   |   |
| 3A4: 0.001375 |   |   |
| 3A5: 0.001275 |   |   |
| 3A6: 0.001174 |   |   |
| 3A7: 0.000974 |   |   |
| 3A8: 0.001475 |   |   |
| 3A9: 0.001375 |   |   |
| 3AA: 0.001275 |   |   |
| 3AB: 0.001475 |   |   |
| 3AC: 0.001275 |   |   |
| 3AD: 0.001375 |   |   |
| 3AE: 0.001375 |   |   |
| 3AF: 0.001375 |   |   |
| 3B0: 0.001475 |   |   |
| 3B1: 0.001174 |   |   |
| 3B2: 0.001174 |   |   |
| 3B3: 0.001275 |   |   |
| 3B4: 0.001275 |   |   |
| 3B5: 0.001275 |   |   |
| 3B6: 0.001275 |   |   |
| 3B7: 0.001375 |   |   |

Repeatability Error, SN = 4130 (Continued)

6-293
| J99 | 0.001275 |  |
| J9A | 0.001275 |  |
| J9B | 0.001475 |  |
| J9C | 0.001275 |  |
| J9D | 0.001275 |  |
| J9E | 0.001174 |  |
| J9F | 0.001275 |  |
| J9G | 0.001275 |  |
| J9H | 0.001174 |  |
| J9I | 0.001174 |  |
| J9J | 0.001174 |  |
| J9K | 0.001275 |  |
| J9L | 0.001275 |  |
| J9M | 0.001275 |  |
| J9N | 0.001275 |  |
| J9O | 0.001275 |  |
| J9P | 0.001275 |  |
| J9Q | 0.001275 |  |
| J9R | 0.001275 |  |
| J9S | 0.001275 |  |
| J9T | 0.001275 |  |
| J9U | 0.001275 |  |
| J9V | 0.001275 |  |
| J9W | 0.001275 |  |
| J9X | 0.001275 |  |
| J9Y | 0.001275 |  |
| J9Z | 0.001275 |  |

The table contains data points with values ranging from 0.001174 to 0.001475. The values are listed in a grid format, with each entry corresponding to a specific location or identifier.

Repeatability Error, SN = 4130 (Continued)
APPENDIX E
BIBLIOGRAPHY

A/D Processing with Microcontrollers, Katausky, Horden, Smith


Analog Devices - Data-Acquisition Databook 1984, Volume 1


Boyces, ed. - Syncro and Resolver Conversion, 1980


Burr-Brown Application Note, Testing of Analog-to-Digital Converters

Burton and Dexter - Microprocessor Systems Handbook, 1977


Electronic Analog-to-Digital Converters, Seitzer, Pretzl, Handy

Handbook of Electronic Calculations, Chapter 15, Analog-Digital Conversion

Harris Analog and Telecom Data Book

IEEE 162

IEEE STD. 746-1984

Intel Application Note AP-124 - High-Speed Digital Servos for Motor Control Using the 2920/21 Signal Processor

Intel Application Note AP-125 - Designing Microcontroller Systems for Electrically Noisy Environments


MIL-M-38510/135 June 4, 1984

MIL-M-38510/135 May 6, 1985

Modern Electronic Circuits Reference Manual

NBS Staff Reports, May/June 1981 P.22/23

Sheingold, ed. - Analog-Digital Conversion Handbook, 1972

Sheingold, ed. - Analog-Digital Conversion Notes, 1977

Sheingold, ed. - Non-Linear Circuits Handbook, 1974

Sheingold, ed. - Transducer Interfacing Handbook, 1980


Distributed Motor Control Using the 80C196KB

TIM SCHAFER
MICHAEL CHEVALIER
80C196KB APPLICATIONS
1.0 INTRODUCTION

Distributed control of servo motors has a wide range of applications including industrial control, factory automation and robotics. The tasks involved in controlling a servo motor include position and velocity measurement, implementation of control algorithms, detection of overrun and stress conditions, and communication back to a central controller. The 80C196KB high performance microcontroller provides a low cost solution for handling these required control tasks.

The 80C196KB microcontroller is a highly integrated and high performance member of the MCS®-96 family. The part is available in ROM (83C196KB) and EPROM (87C196KB) versions. A block diagram of the 80C196KB is shown in Figure 2. The availability of a variety of on board peripherals such as timer/counters, A/D, PWM, Serial Port and High Speed Input and Output capture/compare timer subsystem provides for a flexible architecture for control applications at a reasonable cost.

**Figure 1. Control Tasks for Distributed Control of a Servo Motor**

![Diagram showing control tasks for distributed control of a servo motor](image1)

**Figure 2. 80C196KB Block Diagram**

![Block diagram of 80C196KB microcontroller](image2)
This application note describes several different methods for motor control using the onboard peripherals of the 80C196KB. Hardware and Software techniques are addressed to generate PWMs for driving motors and to measure position from the output of precision optical encoders.

A Proportional, Integral and Differential (PID) algorithm controls both the position and velocity of the motor. The PID algorithm employs proportional, integral and differential feedback to control the system characteristics of the motor. The motor can be moved either manually or under the control of a velocity profile. The mode used to position the motor is determined by commands received from a master controller.

Communication to the master controller was implemented using the onboard serial port of the 80C196KB. The application of distributed control to position and program a six-axis robot arm using six separate motors will be described. Each 80C196KB motor controller acts as a slave under control of the master. An IBM PCTM was selected as the master controller for the robot. Turbo Prolog™ was used to develop the human interface. A robot programming language and control screen was produced to program movements of each individual motor.

The motor control hardware, taking full advantage of the peripheral features of the 80C196KB, will be discussed first. The control software will be discussed later.

2.0 HARDWARE

The hardware tasks required to control a servo motor under the command of a centralized controller include the following:

1) Feedback of the motor position and direction.
2) Control of the motor speed and direction.
3) Detection of motor overrun conditions.
4) Communication from/to a master controller.

Two different hardware interface examples for controlling a servo motor are shown in Figures 3 and 4. The first example controls one motor using TIMER2 and the dedicated PWM unit on the 80C196KB and would best fit a high performance, high resolution application. Example number 2 uses the HSI (High Speed Inputs) and the HSO (High Speed Outputs) to control two motors. The second method can control up to four motors by trading off some performance and resolution.

This section deals with the hardware and software requirements of acquiring position feedback from incremental shaft encoders and generating outputs to drive DC servo motors. A current limiting circuit which is useful in determining when the motor has stalled is also presented. Current monitoring can also control the torque to prevent the motor from crushing an object. The closed loop digital control algorithms are discussed in the software section.

2.1 Optical Encoders

Optical encoders can be used to measure the position of rotating equipment. They provide a cost effective solution for digital position and velocity feedback to a microcontroller. Encoders produce two pulse trains which give an incremental position count. Velocity and acceleration may be calculated by measuring the number of counts in a given sample period. Or, in a slow speed system, velocity and acceleration can be measured directly from the time between edges of the pulse train. Acceleration and velocity calculations are discussed in detail in the software section.
Pulse trains from an encoder can vary from two pulses per revolution for low cost applications, to over 5000 pulses per revolution for high resolution requirements. Figure 5 shows an eight line encoder along with the associated waveforms. A small amount of external logic and a few discrete components decode a position count and a direction indication from phase A and phase B.

External logic for encoders is shown in Figure 6. Figure 7 shows a timing diagram of the circuit. Bold type denotes the input and desired output waveforms. The phases from an encoder are mechanically produced electrical signals. When the motor rotates slowly, the phases inherently exhibit slow rise and fall times. The four Schmitt triggers in the circuit protect against oscillation in the digital circuit due to these long rise and fall times.

Inside track generates Phase A and outside track generates Phase B

Figure 5. Eight Line Encoder
A simple one-shot is constructed with an RC circuit and an XOR gate to generate a pulse on each edge of each phase. ASHOT clocks phase B and BSHOT clocks phase A. This technique of digital filtering insures repetitive edges on a single phase without an edge on the other phase are not passed on to the processor. Repetitive edges occur when the motor changes direction. Further logic obtains a direction or UP/DN bit. Note the first edge after a direction change is lost. A lost edge does not affect the count since the first transition is lost in both directions. Since an edge is lost in each direction, the circuit has an absolute resolution of one edge.
2.2 Interfacing to TIMER2

COUNT indicates an incremental position count on both its rising and falling edge. TIMER2 on the 80C196KB is a 16 bit externally clocked up/down counter clocked on the rising and falling edge of its input signal. A one or zero on port pin 2.6 determines whether TIMER2 counts up or down. By interfacing an optical encoder to TIMER2 as shown in Figure 8, an up/down counter is realized. No software intervention is required to keep track of position or direction changes with the 16 bit TIMER2. The CPU is free to concentrate on executing the control algorithm.

For designs requiring greater resolution, a 32-bit up/down counter may be realized with the same circuit and minimal software overhead. TIMER2 can cause an interrupt on an overflow condition. However, an overflow interrupt is not the safest way to implement a 32-bit up/down counter. Repetitive overflow interrupts could happen when the motor oscillates about a position where the LSW (Least Significant Word) is zero, or TIMER2 keeps overflowing and underflowing. For this method, the total software overhead required for a 32-bit up/down counter is dependent on the position and set point of the motor and would be difficult to predict.

A much better way to implement a 32-bit up/down counter is shown in Figure 9. TIMER2 is only read at the beginning of the control algorithm, or once a sample time. This does not present an accuracy problem for a digital control algorithm. TIMER2 is read into a temporary register. The temporary value is then subtracted from TIMER2, rather than clearing TIMER2, ensuring no counts will be missed. The 16-bit temporary value is sign extended to form a two’s complement 32-bit value and added to the old 32-bit position value to form the current position value. This 32-bit up/down counter provides the accuracy needed for a control loop while keeping software overhead constant under all conditions.

A Pittman motor with a Hewlett Packard HEDS - 5310 512 line incremental shaft encoder was interfaced to TIMER2. Even at a maximum shaft rotation of 6000 RPM, the edges are only clocked into TIMER2 at a period of about 5μs.

\[
(6000 \text{ RPM}) \times (1/60 \text{ M/SEC}) \times (512 \text{ LINE}) \times (4 \text{ EDGES/LINE}) = 204,800 \text{ edges per second}
\]

TIMER2 has a minimum transition period of once a state time, or 167 ns @ 12 MHz, in the fast increment mode. Obviously, much higher resolutions and speeds may be obtained.

2.3 Interfacing to the HSI

The HSI can interface more than one motor to the 80C196KB. COUNT is input into an HSI pin which is configured to recognize events on both the rising and falling edge of its input signal. UP/DN is input to a port pin to determine direction. Up to four motors can be interfaced to the 80C196KB using the four input pins of the HSI. The disadvantage of using the HSI is an ISR (Interrupt Service Routine) must be executed on each edge. Considerable software overhead could occur if edges are clocked into the HSI faster than about one every 150μs.

Two Pittman motors with 2 line encoders were interfaced to the HSI to generate two 32-bit up/down counters as an example. With both motors turning at a maximum velocity of 6000 RPM, an edge will occur every 625μs. The ISR in Figure 10 processes the edges from the encoders and updates the position values and executes in about 15μs @ 12 MHz on the 80C196KB. This still allows 97.6% \((1 - 15/1250)\) of the total processing time to implement control algorithms and other I/O functions.
The HSI approach does add flexibility. Since the HSI records a TIMER1 value with each transition, velocity and acceleration can be calculated on every edge.

### 2.4 Driving a DC Servo Motor

Figure 11 shows the circuit used to drive the motors. A digital output from the 80C196KB is converted into an analog signal capable of driving a DC servo motor. POWER is a PWM output from the 80C196KB. DIRECTION is a port bit which qualifies the +15 or −15 supply. A signal diagram is shown in Figure 12. Isolation between the motor power supply and the digital supply is provided by the two optical isolators preventing any inductive glitches caused by the motor turning on and off from effecting the digital circuit. The optical isolators in turn drive the two VFEₜs. Size of the VFEₜs was determined by the current specifications of the motors. Heat sinks were used to protect the VFEₜs. The VFEₜs are protected from voltage spikes by the MOV, (Metal Oxide Varistor), a type of transient absorber.

### 2.5 Using the Dedicated PWM Output

The PWM output unit on the 80C196KB is an 8 bit counter which increments every state time. The output is driven high when the counter equals zero and driven low when the counter matches the value in the PWM_CONTROL register. Typical PWM waveforms are given in Figure 13. A prescaler can allow the PWM counter to increment every two state times. With a 12 MHz crystal, the PWM has a fixed output frequency of 23.6 KHz, or 11.8 KHz with the prescaler enabled.

---

```assembly
hsi_data_int:
    pushf
    orb iosl_bak,iosl
    jbc iosl_bak,7,no_data
more_in_fifo:
    andb iosl_bak,#0111111b
mot_4_cnt:
    jbc hsi_status,0,mot_5_cnt
    jbs portI,0,mot_4_up
    sub mot_4_pos,#1
    subc mot_4_pos+2,#0
    br mot_5_cnt
mot_4_up:
    add mot_4_pos,#1
    addc mot_4_pos+2,#0
mot_5_cnt:
    jbc hsi_status,4,test_again
    jbs portI,1,mot_5_up
    sub mot_5_pos,#1
    subc mot_5_pos+2,#0
    br test_again
mot_5_up:
    add mot_5_pos,#1
    addc mot_5_pos+2,#0
test_again:
    ld ax,hsi_time
    nop
    nop
    nop
    orb iosl_bak,iosl
    jbs iosl_bak,7,more_in_fifo
no_data:
    popf
    ret
```

**Figure 10. HSI Interrupt Service Routine**

**Figure 11. Circuit Used to Drive the Motors**

**Figure 12. Isolation Circuit**

**Figure 13. Typical PWM Waveforms**
Figure 11. Motor Drive Circuitry

Figure 12. Motor Drive Waveforms

Figure 13. PWM Output Waveforms
The PWM unit along with pin 2.7 was used to drive one motor at a fixed output frequency of 23.6 KHz. By driving the motor at this frequency, motor whine in the audible range was eliminated. Note that a 00 value in the PWM register applies full power to the motor; the desired 8 bit output value must be inverted before it is loaded into the PWM_CONTROL register to obtain the correct output.

2.6 Using the HSO to Generate PWMs

The HSO (High Speed Outputs) of the 80C196KB can generate up to four PWMs. The HSO triggers events at specified times based on TIMER1 or TIMER2. For the specific purpose of generating PWMs, the event is driving an output pin high or low. HSO commands are loaded onto the CAM (Content Addressable Memory), which specify the time and event to take place. The CAM is eight positions deep. The HSO triggers the event on a successful compare with the associated timer.

The 80C196KB can optionally lock commands onto the CAM. This feature is very useful for generating PWMs using TIMER2 as the time base. Figure 14 shows an example of two PWM outputs using locked commands in the CAM. TIMER2 is clocked externally at a frequency which determines the resolution of the PWMs. TIMER2 can be clocked at a maximum frequency of once every eight state times (1.33µs @ 12 Mhz) when used with the HSO. The RESET TIMER2 @ T4 command specifies the output frequency of the PWMs. By changing the external TIMER2 clock frequency and the value of T4, the HSO can generate a wide range of PWMs.

For this ap-note, two Pittman motors were controlled using the HSO along with port pins 2.6 and 2.7. It was desired to keep the output frequency the same as the output frequency of the on-board PWM. To accomplish this, TIMER2 was clocked every 8 state times and reset when it reached 31 counts. This makes the output frequency 23.6 KHz @ 12 Mhz with 5 bits of resolution. CLKOUT was externally divided by 16 and input into TIMER2. Since TIMER2 counts on both the positive and negative edge of its input signal, a square wave with a 16 state period clocks TIMER2 every 8 state times.

The ISR used to load commands onto the CAM is shown in Figure 15. When the control algorithm determines an output has changed, a RESET TIMER2 command gets loaded onto the CAM to generate an interrupt. The interrupt vectors to this routine and updates the CAM. To clear a locked entry from the CAM, the entire CAM must be flushed by setting IOC2.7. Care must be taken to reload all of the commands. This includes any commands not locked on the CAM.
There is the potential for commands to be missed when they are flushed and reloaded on the CAM. For example, an HSO command is loaded on the CAM to clear HSO pin 3 when TIMER2 = 23 and the CAM is flushed when TIMER2 = 22. A new HSO command is then loaded onto the CAM to clear HSO.3 when TIMER2 = 21. This command will not execute until TIMER2 is cleared and counts back up to 21. Missed commands are difficult to avoid without excessive software overhead. Software must take missed commands into account and minimize the effects on the application.

The ISR in Figure 15, insures if an output edge is missed for one period of TIMER2, the HSO pin will remain high. A logical one applies no power to the motor. Also, at the end of the routine a sanity check makes sure TIMER2 is not greater than 31.

### 2.7 Current Limiting

When a motor is stalled, or excessively loaded, it will draw a lot of current. Current limiting can be used to keep the motor from damaging itself, or anything in its path. Several options exist to the user on what to do about a high current condition. Less power could be applied, or the motor could shut off entirely. This section only explains how to recognize a high current condition in a DC servo motor, not what to do about it.

Figure 16 shows a way to convert the current from the motor into a voltage which can be read by the 80C196KB onboard A/D converter. Again, an opto-isolator keeps the motor and digital power supplies separate. When enough current flows through the opto-isolator, the A/D input voltage will drop down to about .7 volts. The current to the opto-isolator is varied by changing the values of the two resistors, R1 and R2 which split the current flow. By changing R1 and R2, this circuit can be adjusted to work properly with different motors and load conditions.
Motor startup current must be considered when testing for a high current condition. When a motor is started, it will draw a great deal of current. This current surge can last for a few milliseconds. Software must decide if the motor is drawing excessive current because it is stalled, or just starting. The section of code in Figure 17 executes during the control algorithm. The current must be above ad_limit for 30 sample times before software recognizes a high current condition. Of course, these values must be adjusted up or down depending on the motor and load conditions.

```assembly
; do a current limit check
jbs ad_command,3,motor_around ; if A/D still running, skip
cmpb ad_result_hi,ad_limit
jh current_ok
incb ad_count
cmpb ad_count,#30
jne current_maybe_ok
; want to do 30h A/D conversions
; before acting because of motor startup current

; here is where the user inserts his code on what to do
; about a high current condition

current_ok:
clrb ad_count

; about a high current condition

current_maybe_ok:
ldb ad_command,#00001001b ; start another a/d conversion

motor_around:
```

Figure 16. Current Sensing Circuitry

Figure 17. Current Sensing Software
3.0 SOFTWARE

A block diagram of the software is shown in Figure 18. The software consists of a main program for hardware and software initialization of the 80C196KB peripherals and programming of control tasks. The control tasks include tracking the motor position and direction, control of the motor speed and direction, detection of overrun conditions and communication to the master controller. After initialization is complete, the 80C196KB enters idle mode to preserve power while not performing control tasks. Interrupt service routines for the serial port, HSI, HSO and software timer perform the various control tasks.

The communication protocol to the main controller is implemented in the serial receive and transmit routines. Commands from the master controller move the motor in one of two modes, manual or automatic, depending on the command. The commands are listed in Figure 28.

Manual mode moves the motor clockwise or counterclockwise with a preset maximum control voltage applied. Manual mode commands include MOTOR UP, MOTOR DOWN and STOP. The MOTOR UP and MOTOR DOWN commands send the motor into manual mode. The motor continues to run in the desired direction until a STOP command is issued from the master controller. The STOP command loads the destination position with the current position and enters automatic mode.

Automatic mode positions the motor using either a position or velocity PID algorithm. The position PID algorithm is applied after reception of the STOP command or when the desired position is reached. The destination position can be changed by a POSITION command from the master controller.

The maximum motor velocity and the destination position are contained in the POSITION command. If the maximum velocity is zero, a position PID is applied to move the motor to the destination position. A non zero maximum velocity will position the motor using a velocity PID algorithm. Position and velocity input to the algorithms are calculated based on position input from the encoder.

Position information for the PID algorithms can be provided by using the High Speed Inputs or TIMER2. The HSI interrupt routine processes the direction and position information incoming from the encoder to provide current motor position. Alternatively, TIMER2 directly measures the position when used as an up/down counter. Velocity information can be calculated using the position information given a constant sampling rate. The position and velocity information are used by the PID control algorithms.

The control algorithm uses a software timer interrupt to generate the sampling rate of the control software. The main portion of the software timer routine calculates the current position and velocity, senses the motor
current for overrun conditions, calls the PID control algorithm and generates the PWM control voltage to the motor.

The speed of the motor can be controlled using the PWM or the HSO. If the HSO is used, the HSO interrupt routine generates a PWM output to control the voltage applied to the motor. Otherwise, the PWM unit controls the voltage applied to the motor.

Each of the major software routines is covered in detail in this section.

### 3.1 Main Initialization Routine

The main initialization routine executes immediately following reset to initialize the 80C196KB peripherals and enable the interrupt driven control tasks. A flow chart for the main initialization routine is shown in Figure 19. The constants and variables for the control algorithms and software routines are loaded into register space for fast execution.

Next, the various peripherals are programmed to handle the control tasks. The PWM for voltage control of the servo motor is initialized. TIMER2 is programmed as an up/down counter with T2CLK as the clock source. The serial port is set to 19.2 Kbaud and programmed for mode 2 to receive incoming commands. An A/D conversion is started to check for initial stress conditions. Before the motor can be accurately positioned, an initial reference point must be established.

In order to find the reference point, an I/O port is connected to a limit switch. The motor is driven in a preset direction until the limit switch is activated. The initial position is then loaded and position PID control is applied to keep the motor stable. Position commands from the master controller can now precisely position the motor from the established reference point.

Finally, the software timer, timer overflow, receive and transmit interrupt routines are enabled and the idle mode is entered to conserve power. The routines will execute as each individual interrupt control task requires servicing. Discussion of the control tasks of each software routine is contained in the following sections.

### 3.2 Software Timer Interrupt Routine

The software timer interrupt service routine executes every 500 µs and determines the sampling rate of the PID control algorithm. Figure 20 shows the flow chart for the software timer interrupt routine. The routine determines the operating mode, calculates the current velocity and position and tests for overrun of preset boundary conditions and stress conditions.
An A/D conversion compares the motor current to test for a stress condition against a preset limit. Thirty conversions are done to average the motor current to prevent a false trigger due to a large current surge when the motor starts up. If the preset limit indicating a stress condition is exceeded, the motor is stopped.

The motor is also stopped if the current position exceeds the preset boundary limits. In the case of the robot, the movement of joints are limited to prevent positions which may cause stress conditions or damage the robot. The positioning of the robot is dependent on the mode of operation.
A manual flag is tested to determine if the automatic or manual mode should position the motor. The manual mode moves the motor either up or down with a preset maximum motor control voltage until a STOP command is issued. The automatic mode positions the motor using either the position PID for accurate positioning or the velocity PID for long positioning.

The software timer interrupt routine calculates and stores the current position and velocity of the motor for use by the appropriate PID algorithm. The current velocity is calculated given the sampling rate, the current position and the previous position. The calculated velocity and position information is stored in the 80C196KB register space for use by the PID algorithm software.

Recall that either a position PID or a velocity PID control algorithm will be executed depending on the maximum velocity value passed by the master controller. If the value is zero, a position PID is employed; otherwise, the velocity profile is employed. The velocity profile PID is ideal for large maneuvers while the position PID is better for shorter movements or maintaining the current position. The generated output from the control algorithm is then loaded into the PWM control register and a return is executed.

3.3 PID Control Algorithm

The algorithm used to control the angular position and velocity of the motor is a common PID algorithm. The algorithm uses proportional, integral and differential feedback to control the output to a motor. The PID algorithm controls the important system characteristics of the motor: settling time, steady state error, and system stability. Each term in the control algorithm affects each system characteristic differently. A block diagram of the PID algorithm is illustrated in Figure 21.

![Figure 21. Block Diagram of PID Algorithm](image)

The PID algorithm consists of three terms: a proportional term, integral term and differential term. The proportional term drives the motor with an output directly proportional to the error between the desired and measured position. The integral term consists of the integral of the position errors multiplied by an integral constant. The differential term is the change in error multiplied by a differential constant. The sum of the terms is then scaled to provide a control voltage to the motor. The system characteristics of the motor are tuned by the selection of appropriate constants.

The settling time, steady state error and system stability are impacted by the amount of proportional gain selected. To accurately control a small change in motor position, a large gain is desired. Faster system response is attained by selecting a large gain but at the cost of greater overshoot and longer settling time. The effect of varying loads on the motor makes proportional control in itself inadequate because of system instability and large steady state error.

Application of integral feedback drives the steady state error to zero by increasing the output in response to a steady state error. The integral term increases as the sum of the steady state error increases causing the error to eventually be driven to zero. The integral term, although driving the steady state error to zero, can cause overshoot and ringing if it is too large. This has the undesirable affect of poorer system response. Applying PI control works very well, however a faster system response can be achieved by applying a PID algorithm.

System response can be improved by adding a differential term. Addition of this term improves the response time by providing a output proportional to the rate of change in error. When the motor has a large change in error, the term produces a large output to the motor. Therefore, the system responds faster to disturbances in the system. Most of the system instability is caused by too high of a differential constant. The size of the proportional, integral and differential constants provide tradeoffs to the desired system characteristics.

Selection of the three gain constants is critical in providing fast system response with good system characteristics. A slightly modified PID algorithm controls the motor which improves both the system response and the system stability. Two modifications were made to improve the control algorithm. First, the size of the integral term was clamped to prevent instability caused by an extremely large integral term which could occur after a long time with large errors. Second, the integral term was cleared when the error changed sign to further improve the system stability. The PID control algorithm is written in PL/M-96 for ease of development.

3.4 Position PID Software

The software flow chart for the PID algorithm is shown in Figure 22. Upon entering the routine, the position
Figure 22. Position PID Algorithm
error is checked for a minimum value before applying the position PID algorithm. If the minimum position error is exceeded, the maximum PWM output is applied to move the motor as rapidly as possible.

Current position error is added to the integral sum. Position error and integral sum are tested to clear the integral sum if they are opposite in sign. This improves the system stability by preventing the integral term from applying a correction opposite to the desired output.

If the integral sum is greater than the maximum sum allowed, the integral sum is clamped. This prevents the integral sum from becoming too large if the error is large for several samples. Differential error is then calculated from the current and previous position errors.

Output for the PID algorithm is calculated from the proportional, integral and differential terms multiplied by their individual gain constants. The output is then scaled and tested for the preset PWM output limit. If the limit is exceeded, the output to the PWM is set to the maximum value. The appropriate motor direction is set depending on the sign of the output. The final output to the PWM control is ready and the software returns.

3.5 Velocity Profile

Positioning of a servo motor using only a position PID algorithm wastes power and gives poor system performance when moving between two positions. A velocity profile provides a smooth transition between two angular positions and improves the energy consumption of the motor. Three different velocity profiles which can be applied are trapezoidal, triangular and parabolic.

The parabolic profile is the most power efficient and provides smooth acceleration and deceleration at the end points. However, a large amount of processor time is needed to calculate the profile in real time. The triangular profile provides ease of calculation and relatively smooth acceleration and deceleration throughout the velocity profile. For these reasons, the trapezoidal profile was selected.

A trapezoidal profile consists of an acceleration period, run period and deceleration period. The variables ACCEL_TIME, RUN_TIME and END_TIME represent the periods. Figure 23 shows the trapezoidal profile. Acceleration and deceleration rates for the motor are fixed according to the optimum values found through testing. The master controller sends a position command containing the maximum velocity (MAX VELOCITY) and the desired end position (DES POSITION). The DES POSITION is equal to the integral of the velocity profile (i.e., the final position can be determined by integrating the velocity over the period of the profile). Therefore, the ACCEL_TIME, RUN_TIME and END_TIME can be calculated based on the DES POSITION, ACCELERATION, DECELERATION and MAX VELOCITY.

The destination position should be reached if the velocity profile was ideally tracked. However, a certain amount of position error can be expected as the motor travels from one point to another. This error is eliminated by applying the position PID at the end of the velocity profile. This modified control algorithm has both good motor performance and accurate angular positioning.

![Figure 23. Trapezoidal Velocity Profile](image-url)
3.6 Trapezoidal Profile Calculation

The trapezoidal velocity profile is calculated when a position command with a nonzero maximum velocity is passed from the master controller. The master passes the desired end position and the maximum velocity of the motor. A reasonable acceleration (deceleration) rate was found through experimentation to be 1 position count/sampling rate (500 μs). ACCEL_TIME, RUN_TIME and END_TIME can be easily calculated given the relative acceleration rate of one, the end position and the maximum velocity.

The acceleration and deceleration time is equal to the maximum velocity since the acceleration/deceleration rate is one. RUN_TIME is the difference between the desired position and current position minus the distance covered during the acceleration and deceleration times. END_TIME is the RUN_TIME added to two times the ACCEL_TIME. With the velocity profile calculated, the velocity PID algorithm will be applied until the END_TIME is reached.

The velocity profile software generates the appropriate velocity depending on the current time. Figure 24 shows the velocity profile generation software. The TIME variable is incremented every software timer interrupt at the sampling rate if it is less than the end time (END_TIME) of the profile. Three different velocities are calculated during the profile. DES_VELOCITY equals the ACCELERATION multiplied by the TIME until the ACCEL_TIME is reached. The DES_VELOCITY equals the maximum velocity until the RUN_TIME is exceeded. Once the RUN_TIME is exceeded, the velocity is equal to the ACCELERATION (same as deceleration rate) multiplied by the TIME-CURR_TIME. When the end of the profile is reached (which is approximately the desired end position), the time equals the END_TIME and the position PID controls the motor. If the maximum velocity passed by the master controller is zero, the CURRENT_TIME is set to the END_TIME and the position PID controls the motor.

The velocity control algorithm employs the PID algorithm. The algorithm is similar to the position algorithm used to control the position. The velocity control algorithm is shown in Figure 25.

![Figure 24. Velocity Profile Generation Software](image-url)
3.7 Fast Execution of Control Algorithms

The high speed arithmetic operations capability, availability of three operand instructions and large register space of the 80C196KB provide for fast execution of control algorithms. The 80C196KB running at 12 Mhz can execute a $16 \times 16$ Multiply in 2.3 $\mu$s and 32/16 divide in 4.0 $\mu$s. Three operand instructions operate on two variables without modification and store the result in the third variable. This eliminates the need for executing load and store operations as required by accumulator bound architectures. The large register space can store all of the constants and variables for the control algorithm without the use of load and store operations. In addition, procedures do not need to pass parameters or store results since they can permanently reside in register space.

A summary of the execution times for the main software routines is shown in Figure 26.

<table>
<thead>
<tr>
<th>Function</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Timer Interrupt Routine</td>
<td>40 $\mu$s</td>
</tr>
<tr>
<td>PID Control Algorithms:</td>
<td></td>
</tr>
<tr>
<td>Velocity PID (PL/M-96/ASM-96)</td>
<td>300 $\mu$s/30 $\mu$s</td>
</tr>
<tr>
<td>Position PID (PL/M-96/ASM-96)</td>
<td>240 $\mu$s/40 $\mu$s</td>
</tr>
<tr>
<td>Velocity Profile Generation</td>
<td>71 $\mu$s</td>
</tr>
<tr>
<td>HSI Interrupt Processing</td>
<td>22 $\mu$s</td>
</tr>
<tr>
<td>HSO Generate PWM Routine</td>
<td>16 $\mu$s</td>
</tr>
<tr>
<td>Receive Interrupt and Command Processing</td>
<td>26 $\mu$s</td>
</tr>
<tr>
<td>Transmit Interrupt Routine</td>
<td>11 $\mu$s</td>
</tr>
</tbody>
</table>

Figure 26. Execution Times for Main Software Routines
The HSI, HSO, Receive and Transmit Interrupt routines take a minimal amount of time. A majority of the processing time is in executing the Software Timer interrupt routine and either the Velocity PID or Position PID control algorithms.

PID Control Algorithms take a considerable amount of time since they are written in a high level language and execute a number of thirty-two bit arithmetic operations. Thirty-two bit accuracy is not required since the maximum position required to accurately track the motor is about twenty four bits. To optimize the control algorithm for the accuracy required, the routines can be written in assembly. A sample Position PID algorithm is shown in Figure 27. The routine executes in about 30 μs by optimizing the control algorithm and minimizing the number of 32-bit operations.

```
VPID:
    ld vel_err3,vel_err2 ; store velocity errors
    ld vel_err2,vel_err1
    ld vel_err1,vel_err
    sub vel_err,des_velocity,velocity ; calculate velocity error

    sub temp,vel_err1,vel_err2 ; calculate differential error term
    mul temp,#3
    ; diff_err=(vel_err1-vel_err2+3*vel_err1-3*vel_err2)
    sub temp,vel_err3
    add temp,vel_err

    ; Output=prev_output + ((vel_err1-vel_err2)*VKp+(Vel_err1+Vel_err2)*Vki + diff_err*Vkd)/;scaler
    mul temp,Vkd
    add temp2,vel_err1,vel_err1
    mul temp2,Vki
    add temp,temp2
    sub temp2,vel_err1,vel_err1
    mul temp2,Vkp
    add temp,temp2
    div temp,scaler
    add output,prev_output,temp
    ld prev_output,output
    div Out_scaler ; scale output
    jbc Out+3.7,forward ; Scale 32 bit output to get 16 bit result
    neg Out+2 ; test output for direction
    sjmp scaleout
    ldb p2,#07fh ; set direction down(p2.0=0)
    ldb p2,#OFFh ; set direction up(P2.0=1)

REVERSE:
    cmp Out,#0ffh ; scale output for maximum pwm value
    jgt exit ; if out > maximum pwm output
    ldb Out,#0ffh ; then clamp output to max pwm value

FORWARD:
    ldb p2,#OFFh

SCALEOUT:
    cmp Out,#0ffh
    jgt exit
    ldb Out,#0ffh

EXIT:
    ldb pwm,Out
    ret
```

Figure 27. Position and Velocity PID Assembly Routine
4.0 Distributed Control

Distributed control of servo motors requires the passing of commands and data from a master to a slave. The master passes commands to report position, start and stop the motor, or position the motor to an exact location using a position PID or velocity profile. The slave needs to report current position and acknowledge incoming commands from the master. This protocol requires addressing of slaves and the distinction between incoming commands and transmission of data. The 80C196KB serial port provides a multiprocessor communication mode for implementing the protocol.

The 80C196KB provides a ninth bit in Mode 2 and Mode 3 that can assist communication between multiple processors. If the received ninth bit is zero in mode 2, the serial port interrupt will not occur. Each motor is initially programmed for this mode to distinguish receiving a command versus a data byte. With the ninth bit set, indicating a command byte has been received, all the slaves interrupt and process the incoming byte. The address of the motor being controlled is embedded in the command byte. All processors will process the command byte if the motor address matches.

A motor receiving a poll command from the master controller will enter mode 3. The polled motor then receives the data bytes which are sent with the ninth bit cleared. Therefore, only the processor receiving data will interrupt for serial reception while the other processors awaits another command byte with the ninth bit set. A list of available commands and the format for each is shown in Figure 28.
Command Table

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>01</td>
<td>Position motor using either position PID or Velocity profile.</td>
</tr>
<tr>
<td>Poll</td>
<td>05</td>
<td>Polls motor for current position.</td>
</tr>
<tr>
<td>Motor Up</td>
<td>08</td>
<td>Enters manual mode turning motor clockwise.</td>
</tr>
<tr>
<td>Motor Down</td>
<td>09</td>
<td>Enters manual mode turning motor counter clockwise.</td>
</tr>
<tr>
<td>Stop</td>
<td>10</td>
<td>Exits manual mode setting the desired position to the current position.</td>
</tr>
</tbody>
</table>

Position Command

<table>
<thead>
<tr>
<th>Command</th>
<th>Position</th>
<th>Maximum Velocity</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>4 bytes</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

Poll Command

<table>
<thead>
<tr>
<th>Command</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Figure 28. Master Commands and Format

4.1 Receive Interrupt Service Routine

Communication between the 80C196KB and the main controller is handled by the serial port routine. Figure 29 shows the flow for the receive interrupt service routine. Upon reception of a byte from the main controller, a receive interrupt will occur. The RI bit is tested to ensure a byte has been received. If a byte has not been received, an error is generated and a return from the routine is executed. After a valid reception, the ninth bit is tested to determine if the incoming byte is a command byte or incoming data sent after reception of a POSITION command.

If the byte is a command byte, the motor address is checked by each slave for its own address. The command byte is then echoed back to the master controller by the appropriate slave. The routine is exited if the command byte is not for the motor. Since each motor has a unique address, only the motor receiving the command will respond. Reception of a POSITION command will switch the serial port to mode 3.

Desired position and maximum velocity is sent by the master to each slave by a POSITION command. Received data for the position command is stored in a buffer. After all data has been received, MAX_VELOCITY and DES_POSITION is loaded with the values stored in the buffer and the serial port is switched back to mode 2.

Each command is then checked and appropriate action taken depending on the received command. Commands include POSITION, POLL, UP MOTOR, MOTOR DOWN and STOP. The commands are summarized in Figure 28.

4.2 Manual Positioning

The receive routine will check for one of three manual commands: MOTOR UP, DOWN MOTOR or STOP. A manual flag is used by the software determine if the motor should be positioned using either a position or velocity PID algorithm or by manual control. The motor up and motor down commands set the manual flag which will cause the PWM control to be loaded with a constant value during the software interrupt routine. The direction port bit is set to the appropriate value depending on whether the command is up or down. The motor will continue to move up or down until a STOP command is issued by the master controller or the motor's preset limits are reached.

A stop command will reset the manual flag and set the controller in automatic mode which employs the PID algorithm. The destination position gets loaded with the current position and a return from the receive interrupt is executed. The manual position mode is used by the master controller to position the motor under keyboard or switch control. This is instead to precise position control of the motor by sending a position command.

4.3 Motor Positioning

Either position control or a velocity profile can be used to position each motor. The maximum velocity information stored in the POSITION command determines the type of method employed. If the maximum velocity value is nonzero, the velocity PID algorithm will be applied to position the motor. If the maximum velocity is zero, position control using the PID algorithm will be used. This provides for two alternative methods for positioning the motor.
Once a POSITION command is received, the processor enters serial mode 3 to receive the incoming position and maximum velocity information. The four bytes of position data and two bytes of maximum velocity are retrieved from a six byte storage buffer. A receive count keeps track of the number of incoming bytes until all bytes of the six byte frame have been received. If a frame or overrun error occurs, the motor will shut off and a 0FFH will be transmitted back to the master controller to indicate an error condition has occurred. Otherwise, an 88 is returned to indicate the valid transmission of position and maximum velocity. The manual flag will be turned off and the appropriate PID algorithm will be applied on the next software interrupt.

4.4 Master Polling of Position

The master controller can poll each motor controller for position with a poll command. After reception of the poll command, a transmit buffer is loaded with four bytes of position information. Each byte is then transmitted using the transmit interrupt routine.

The flowchart for the routine is shown in Figure 30. The routine simply tests the TI flag and continues to transmit a byte from the buffer until the transmit count goes to zero. After the count goes to zero, the transmission is complete and processing continues.

![Figure 29. Serial Port Receive Interrupt Routine](image-url)
Figure 29. Serial Port Receive Interrupt Routine (Continued)
5.0 DISTRIBUTED CONTROL OF A SIX AXIS ROBOT

A six axis robot demonstration system was built using distributed control of its six motors. The robot is a RHINO XR-1™ prototype robot designed by SANDHU Machine Design Inc. Robot motors were replaced with similar models with high resolution encoders. The robot allows movement along six joints: base, shoulder, elbow, wrist, hand and fingers. Each joint is connected to a motor. The system used an IBM PC acting as a master controller.

The software used to develop the human interface was Turbo Prolog and the Turbo Prolog Toolbox. The human interface allowed for the programming and movement of the robot by individually controlling each joint motor. The IBM PC controlled each axis of the robot by passing commands serially.

The IBM PC provides a flexible master controller for positioning the robot. There are a large number of software languages for developing the control algorithms and human interface of the master controller. Turbo Prolog was selected for its low cost and ease of implementation. The control screen and robot programming language were rapidly developed using the Turbo Prolog. The software and hardware implementation easily provide for programming and controlling the robot through a variety of repetitive tasks. A robot using this control system could easily perform assembly or manufacturing tasks as shown in Figure 31.

5.1 Hardware Interface

The hardware interface to the robot is shown in Figure 32. Each major joint, elbow wrist, base and shoulder were controlled with a single 80C196KB using the PWM and TIMER2 as an up/down counter. The hand and finger motors used the HSI to track position and the HSO to generate PWM motor control voltages.
Figure 32. Robot Control Hardware Block Diagram
Switches on the robot were fed into 80C196KB I/O ports to provide a reference position when each motor starts up. Current sensing for each motor was fed back to the analog channels to provide an indication of any overrun or stress conditions. Limits were set for each motor to prevent the robot joints from entering positions where obstacles or mechanical limitations were reached.

Each motor was given a unique programming address for communication back to the master controller. The master controller sent commands with the address of whichever joint motor needed to be positioned or polled. The master 80C196KB communicated through a UART to the IBM-PC.

5.2 Human Interface

To control the robot, the human interface provided a variety of programming options.

The software features included:

- Manual control via the keyboard
- Editing robot command files
- A Motor Control Command language
- Table Display of motor position and status
- Manual Programming mode
- Table Positioning mode

The software front end developed only the basic features of robotic control to demonstrate the distributed control of servo motors.

5.3 Control Screen for the Robot

The screen for the control of the robot is shown in Figure 33. The screen displays a table of the position and status of each motor, shows the function keys used to execute commands or enter different modes and displays the keyboard keys for moving each robot joint up or down. The software has various modes for positioning and programming the robot.

5.4 Programmed Modes

The software provides for movement of the robot through table entry, execution of include command files or manually using the keyboard. The robot is positioned manually by entering the function key for manual mode and then pressing the predefined key for each joint motor to move up or down. As each key is released, a STOP command is issued to each motor. The motors are then polled and the current position updated in the table.

The table function allows for direct entry of the desired position and maximum velocity to position the motor when the table function key (F1) is pressed. After the

<table>
<thead>
<tr>
<th>Motor</th>
<th>Position</th>
<th>Maxval</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>12345</td>
<td>0</td>
<td>STOPPED</td>
</tr>
<tr>
<td>Shoulder</td>
<td>13457</td>
<td>0</td>
<td>STOPPED</td>
</tr>
<tr>
<td>Elbow</td>
<td>00282</td>
<td>0</td>
<td>STOPPED</td>
</tr>
<tr>
<td>Wrist</td>
<td>00383</td>
<td>0</td>
<td>STOPPED</td>
</tr>
<tr>
<td>Hand</td>
<td>11228</td>
<td>0</td>
<td>STOPPED</td>
</tr>
<tr>
<td>Fingers</td>
<td>18484</td>
<td>0</td>
<td>STOPPED</td>
</tr>
</tbody>
</table>

Figure 33. Robot Control Screen
key is pressed, individual positioning commands are sent to each motor. With maximum velocity set to zero, the motor is positioned using a position PID. A non-zero maximum velocity would position the motor using a velocity profile. The final method of positioning allowed for the sending of commands (MOTOR UP, MOTOR DOWN, STOP, POSITION or POLL) to each joint in the robot from an include file.

The include mode function key (F6) executes commands stored in a file. The command file can be entered using an external editor or using the on board editor, Turbo Prolog. A sample command file is shown in Figure 34. The command file allows for programming of the robot through a sequence of programmed tasks. The task of programming the robot is eased by a manual program mode.

The manual program mode generates a command file while manually positioning the robot. After pressing the program key (F4), the program mode is entered and the robot is moved by pressing the appropriate motion key for each joint motor. When the robot stops, the position of the robot is polled and translated into a position command and stored in a file. As the programmed task is executed, each position of the robot and the time delay between joint movements is recorded. When the task is complete, the file contains all the stored position commands necessary to execute the programmed task. The file can be edited with by entering the edit mode (F5) to fine tune the programmed task or execute the command file directly. The manual program, command file execution and editing modes allow for a variety of robotic tasks to be developed and tested easily.

6.0 CONCLUSION

Use of an 80C196KB in distributed control of servo motors has been demonstrated with the effective utilization of the onboard peripherals and high speed math capability of the 80C196KB. The high performance and integration of the 80C196KB minimized the hardware interface. The task of controlling the motor resided in the 80C196KB with the control algorithm residing in the master. With this approach, the centralized controller can be adapted to the performance requirements of the system.

Although not implemented, a learn mode could be added to the robot to provide programming using AI techniques. The IBM PC and Turbo Prolog software provided the demonstration vehicle for testing the control of the robot using distributed control. Use of artificial intelligence programming to position the robot could be incorporated with the Turbo Prolog package. The application of a vision system or a more complex control algorithm could be realized without modification to the hardware controlling the robot. A more cost effective solution is obtained by replacing the IBM-PC with one 80C196KB or 80C186 acting as a master controller.

Repetitive tasks programmed using the robot command language could be stored in tables in the master 80C196KB. The controller would send the stored commands to each motor and communicate, through a serial UART, to the rest of the manufacturing system. The master 80C196KB controller would then report status or receive commands. The choice of controller depends on the needs of the system. Distributed control of servo motors using the 80C196KB provides for maximum flexibility in the selection of the control algorithm without modification to the hardware control modules.

<table>
<thead>
<tr>
<th>pos(3,4000,10)</th>
<th>move elbow to position 4000 with maximum velocity of 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>time(10)</td>
<td>delay 10 seconds</td>
</tr>
<tr>
<td>pos(1,1000,2)</td>
<td>move shoulder to position 1000 with maximum velocity of 2</td>
</tr>
<tr>
<td>time(20)</td>
<td>delay 20 seconds</td>
</tr>
<tr>
<td>pos(0,14000,5)</td>
<td>move base to position 14000 with maximum velocity of 5</td>
</tr>
</tbody>
</table>

Figure 34. Sample Robot Command File
REFERENCES

Many applications have throughput time requirements on the order of a few hundred milliseconds, and don't require real-time image analysis.

A Single-Chip Image Processor

A.L. Pai and S.H. Lin, Arizona State University, and David P. Ryan, Intel Corporation

Most of the research efforts on image processing focus on expanding the complexity and dimension of image analysis. Unfortunately, this emphasis results in algorithms that are so computationally intensive that expensive special-purpose vector and pipeline processors are required to evaluate an image fast enough to be considered "real-time." Not all applications, however, have the burdensome requirements of true real-life image analysis. Specifically, applications that have image throughput time requirements of greater than a few hundred milliseconds can use a lower cost, general-purpose microprocessor-based system. Applications that have even slower frame rates are candidates for not only the use of lower cost CPUs, but also allow for replacement of video-rate flash A/D converters with slower, less expensive converters.

Addressing the most cost-sensitive applications, the design described herein uses Intel's 16-bit microcontroller to implement a single-chip image processor. The on-chip 10-bit A/D converter of the controller digitizes the image of a charge injection device (CID) camera, while the chip's 16-bit CPU executes a library of standard vision algorithms and reports the results by passing a few tokens over an on-chip universal asynchronous receiver-transmitter (UART).

SYSTEM OVERVIEW

A block diagram of the single-chip image processor is shown in Figure 1. The image is acquired by the CID camera and input as an analog voltage to the 8096 where it is digitized and stored in one of two image buffers. The digital image is stored as an N x N matrix of 8-bit values corresponding to the gray level intensity at each picture element (pixel) as shown in Figure 2.

After an image resides in an image buffer, the 8096 can execute a number of standard image processing algorithms available as system monitor commands. These programs perform thresholding and filtering functions on the digital image, and can analyze objects found within the image. If the 8096 were attached to a host system instead of a terminal, custom programs could be downloaded to the user program RAM and executed.

To view the raw and processed images, a CRT controller is used to keep a video monitor updated with the images stored in the two frame buffer memories of Figure 1. The 8096 updates the frame buffers with the data in its image buffers depending upon commands given to the system.

Hardware. The system is composed of a 128 x 128 CID camera and Intel's 8096 (with on-chip A/D) for image acquisition and analysis. A standard CRT controller was added for displaying raw and processed images as directed by the 8096. Driving the decision to use a 128 by 128 digital
the result. This provides 256 gray levels, and greatly simplifies memory requirements.

Software. In addition to the code necessary to digitize images, the system EPROM contains an extensive set of algorithms for digital image acquisition and analysis. Video operations are used to acquire a digitized image. Point and arithmetic operations involve the pixel-by-pixel manipulation of a digital image. Neighborhood operations produce an output image that is the result of a combination of the gray level intensities around a specified neighborhood of each pixel. Measurement operations include the computation of desired parameters of objects located in an image for pattern recognition and other applications. Finally, utility operations are necessary for system operation.

The algorithms present in the system monitor can be used to identify desired objects in a digital image by following the approach shown in Figure 3. Once an image is digitized, it can be enhanced by the application of various image processing techniques including histogramming, thresholding, and spatial filtering to delineate the desired objects. The 8-directional chain code (Figure 4) can then be used to trace the boundaries of objects, and relevant object parameters can be determined and compared with those of a known object database to identify the unknown object.

OBJECT CLASSIFICATION

In the following example, the 8096 performs a binary thresholding operation as described earlier to set the image background to pure white and the objects in the image to pure black. Then the 8096 searches the image for objects. When an object is found, the object boundary is traced and shape analysis is performed. Descriptive information about the object (or objects) is output over the on-chip UART of the 8096 to a terminal, or host computer. The controller, without consulting a host computer, can also be programmed to make the decision to accept or reject an object on a set of prescribed rules.

The sequence of processing for this example, from serial communication reception and interpretation to the reporting of the shape analysis results, takes approximately 1500 ms with an 8096 running at 12 MHz. The time will vary with the size and number of objects in the field of view.
Photos 1 through 4 show the original 256 gray level digitized image and resultant binary (two-level) image of a circular object and a square object. (The circle looks like an oval when displayed due to the aspect ratio of the video monitor).

Table 1 summarizes the output of the systems shape analysis program. The objects' perimeter (P), area (A), center of mass coordinates (cx, cy), and the coordinates of the endpoints of each minimum enclosing rectangle are listed.

The rectangularity and circularity of the objects were also calculated and appear in Table 1. The rectangularity of the circle and the square using the actual data were ideal. Although the circularity of the digitized circle is slightly different from ideal (12.416 vs. 12.56), the digitized circle can be distinguished from the digitized square since the circularity of the square is very different from a perfect circle (15.44 vs. 12.56).

From these typical results, it is clear that this image processor can be used to distinguish between and identify objects placed in its field of view.

CONCLUSIONS

If the stringent requirements of “real-time” image processing can be relaxed in favor of substantially reduced system cost, a standard 16-bit microcontroller can perform as a stand-alone image processor.

Not only does the design described here demonstrate that a microcontroller can undertake two-dimensional image processing, but the surprising speed with which it accomplishes the processing should lead to the reevaluation of current microprocessor applications for possible cost reduction via microcontrollers.

REFERENCES

Image Processing Techniques

A histogram gives the distribution of all the gray levels in a digital image. The image histogram is used to select a desired threshold intensity level for separating an object from the background in the digital image.

A digital image can be thresholded using various threshold functions (three of which are shown in the figure) to yield an output image that contains a better definition of an object. For example, a binary (black and white) image is obtained by applying the two-level threshold function shown in (c).

\[ g(x,y) = h[f(x,y)] = \sum W_i f(x-x_i, y-y_i) \]

Various types of filter masks can be used to perform different digital image enhancement operations. A low-pass filter uses neighborhood averaging to "smooth" the digital image to remove noisy pixels. A high-pass filter accentuates noisy pixels. A high-pass filter accentuates the higher frequencies present in an image, thus "sharpening" its edges. Operators such as the Sobel masks can be used to compute the gradient at each point in an image, thus producing a gradient edge-detectected image.

Using such filtering methods, the boundaries of objects in an image can be isolated, thus permitting the computation of useful object parameters for object identification and classification.

EXAMPLE SHAPE ANALYSIS OUTPUT

<table>
<thead>
<tr>
<th>OBJECT</th>
<th>PERIMETER</th>
<th>AREA</th>
<th>C.O.M. COORDINATES</th>
<th>ENCLOSING RECTANGLE</th>
<th>RECTANGULARITY</th>
<th>CIRCULARITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIRCLE</td>
<td>301</td>
<td>7297</td>
<td>(62.68)</td>
<td>111 15 116 21</td>
<td>0.785</td>
<td>12.416</td>
</tr>
<tr>
<td>SQUARE</td>
<td>328</td>
<td>6967</td>
<td>(55.73)</td>
<td>97 14 115 32</td>
<td>1.0</td>
<td>15.440</td>
</tr>
</tbody>
</table>

Table 1. Example shape analysis output.

**Size Parameters**

The horizontal and vertical extent of an object and its minimum enclosing rectangle are easily computed by using the minimum and maximum line and sample numbers.

The perimeter (circumferential distance) around an object boundary is obtainable from the boundary chain code by using the formula:

\[ P = N_E + \sqrt{2} N_O \]

where \( N_E \) and \( N_O \) are the number of even and odd steps in the object boundary chain code.

The area of an object, which is a convenient measure of object size, is equal to the number of picture elements inside and including its boundary, multiplied by the area of a single pixel.

**Shape Parameters**

In addition to size parameters, shape parameters can be used to distinguish objects. Some shape parameters that are easily computed are described below.

The formula for computing the rectangularity \( R \) of an object is:

\[ R = \frac{A_O}{A_R} \]

where \( A_O \) is the object area and \( A_R \) is the area of its minimum enclosing rectangle. \( R \) ranges from 0 to 1, with a value of 1.0 for rectangular objects, \( \pi/4 \) (0.785) for circular objects, and smaller values for slender, curved objects.

The aspect ratio, \( A \), which is the ratio of width to length of the minimum enclosing rectangle of an object, is used to distinguish slender objects from roughly square or circular objects.

One of the commonly used circularity measures is:

\[ C = \frac{P^2}{A} \]

the ratio of the square of the object perimeter to its area, which reflects the complexity of the object boundary. \( C \) has a minimum value of \( 4 \pi (12.56) \) for a circular object, while more complex shapes have higher values.

---

A.L. Pai is an Associate Professor and S.H. Lin is a Ph.D. candidate in the Computer Science Dept., College of Engineering, Arizona State University, Tempe, AZ 85281. David P. Ryan is a Senior Application Engineer for Intel Corp., 5000 W. Chandler Blvd., Chandler, AZ 85226.
THE MCS®-96 DIAGNOSTICS LIBRARY

Version X1.1

David Ryan
INTEL Corporation
October 1987
1.0 INTRODUCTION

In the real time world of microcontroller applications, system failures can be dangerous, and expensive. Preventing them, and understanding them when they occur, is very important to the reliability of any design.

The sources of a system upset are varied. But in general, the failure of a well designed application occurs as a result of either some form of noise, or a hardware failure. The 8096 hardware provides methods of detecting and recovering from the transient noise failures, while the MCS®-96 Diagnostics Library supplies software routines that can help diagnose or detect a failure in system hardware.

Graceful recovery from noise induced failures is possible using the WATCHDOG TIMER. While the 8096-based system is functioning as desired, the executing software periodically resets the WATCHDOG with a special two-byte code. If the WATCHDOG is not reset at least every 16 ms (12 MHz system), a system reset occurs. The two-byte code is a unique password which appears nowhere in the opcode map. This reduces the chance that an erroneous WATCHDOG reset would occur after a system upset.

The 8096 RESET instruction provides another form of protection. Since the opcode for a RESET is 0fH, protection against the 8096 executing unimplemented external memory is obtained by placing pull-ups on the system bus. The RESET opcode is also the value in erased EPROMs. Therefore, any attempt to execute non-existent memory or an erased EPROM location causes the 8096 to execute the RESET instruction. RESET causes the 8096 to reinitialize itself and provide an external pulse on the RESET pin to reinitialize the system.

Even with the protection afforded by the 8096, a system is rarely complete without checks for hardware failures, both internal and external to the microcontroller. These checks are usually software routines that execute on power-up or periodically to verify that all parts of the system are present and function correctly. The tests generally execute standard check algorithms which are simply re-written in the host’s assembly language.

To eliminate the need for every designer of an 8096-based system to write such tests, a collection of modular routines has been developed that any designer could easily use in his system (General Diagnostics). In addition, a set of 8096 interrupt service routines was developed for testing 8096 I/O units in a dedicated environment (The Dynamic Stability Test). Both sets of programs are contained in the MCS-96 Diagnostics Library (DIAG96.LIB).

This library is a collection of software modules that provide a number of ready-made General Diagnostics and a specialized MCS-96 diagnostic known as the Dynamic Stability Test. The General Diagnostics implement frequently used standard test algorithms, while the Dynamic Stability Test exercises hardware specific to the 8096.

The library can be considered a software “tool box” from which modules are selected for a variety of run-time diagnostics or laboratory tasks, for example:

- Include a few modules in other programs as a power-up test
- Use a memory module to create a map of external memory
- Use a few modules as a periodic system check
- Develop a simple stand-alone tester
- Build a custom test bed for burn-in, inspection or reliability tests
- Test new background code in an interrupt intensive environment
In addition to easing the development of a program that must perform standard diagnostics or system checks, the library can be a learning tool. Using the proven source code in the library, methods of interrupt management and on-chip peripheral handling can be reviewed to further understand how to use the 8096.

These tests were developed by the 8096 Applications group for experimental use with the 8096. With the programs in this library, the chip has been studied for its functional and asychronous characteristics.

The General Diagnostics should be useful to almost anyone designing an 8096 application. The Dynamic Stability Test will be useful to those experimenting with the 8096 in a test environment. Figure 1 shows the modules in the MCS-96 Diagnostics Library.

### 1.1 General Diagnostics

The General Purpose Diagnostics consist of 24 programs providing System, ALU and Memory tests. Each of the tests can be called independently, and none require special hardware or impose application limiting constraints.

Two Collected Test programs are also provided so that all tests may be called at once. A third Collected Test program executes a selection of General Diagnostics that might be reasonable to include in a typical system power-up.

Section 3 provides a detailed description of the General Diagnostics.

### 1.2 Dynamic Stability Test

The Dynamic Stability Test is an integrated set of 11 programs that provide the interrupt service routines necessary to run all forms of MCS-96 I/O concurrently while a user written main task is executing. Virtually all of the chip is made to run simultaneously, with the I/O units responding to asychronous external events.

Unlike the General Diagnostics, the Dynamic Stability Test modules must all be linked together, and must run in a specific external environment.

Section 4 provides a detailed description of the Dynamic Stability Test.

---

**Figure 1. The MCS®-96 Diagnostic Library**
1.3 How To Use This Manual

This publication is meant to be a guide for those using any of the programs in the MCS-96 Diagnostics Library. On a first pass the entire manual should be skimmed, with more attention paid to Section 2 and the overview sections of Sections 3 & 4. For the casual reader, the overview sections of each chapter should suffice.

Section 2 contains an overview of the general calling conventions to use any test in DIAG96.LIB. The section also describes DIAG96.LIB error reporting conventions and presents some warnings to heed when using this library.

Section 3 describes the classes of General Diagnostics and each test in detail.

Section 4 describes the concept of Dynamic Stability and its implementation on the 8096. The section also contains an overview of the tests performed, a description of the constraints placed upon the user-written background task, and detailed descriptions of each interrupt service routine.

The Appendices contain error code and command file descriptions, and of demonstration program listings. Source for the MCS-96 Diagnostics Library can be obtained from Insite User's Program Library at the address below. The Insite Catalog order number is AE-17.

Insite User's Program Library
INTEL Corporation DV2-24
2402 W. Beardsley Road
Phoenix, Arizona 85027

With the first-hand knowledge that many problems result from not being able to uncover information lodged in some dark corner of the user manual, information is repeated in the sections where it is pertinent.
2.0 USING THE LIBRARY

To simplify use of the diagnostics, the tests were developed in a modular fashion and collected in one linkable object file library (DIAG96.LIB). A modular program relies upon only the parameters sent at its invocation and employs standard parameter passing conventions to allow flexibility and uniformity of use. Collecting the modules into a library eliminates the tedium of listing twenty or thirty file names when performing a relocate/link on user developed code. When a program is linked to DIAG96.LIB, only those modules referenced in the user program are drawn from the library for inclusion in the output module.

Since PLM96 conventions were the ones chosen for this set of programs, the General Diagnostics are invoked by following the conventions for a PLM96 typed procedure. Parameters are placed on the STACK and the procedure activated via a function reference or explicit CALL. When the test is complete, test data is returned in the special register PLMSREG. The Dynamic Stability Test is not PLM96 compatible.

The next section describes the format of the test data that is returned by the diagnostics. Following sections give an overview of how to use a General Diagnostics test, how to use The Dynamic Stability Test, and what restrictions to keep in mind while using the library.

2.1 Reporting Convention

All DIAG96.LIB tests use the PLMSREG word locations ICH and 1EH for returning condition codes to the calling program. Within DIAG96.LIB, these locations are the PUBLIC words EREG1 and EREG2. When a test concludes without finding an error, a zero is placed in the high byte of EREG1. If the high byte of EREG1 is non-zero, then some unexpected condition occurred. The low byte of EREG1 always contains the module number of the returning test, and EREG2 contains a detail code if an error was found. The complete listing of EREG1 code meanings and EREG2 meanings is in Appendices A & B.

All modules cease execution upon detection of the first error. The code describing which error was detected (EREG1) follows the format described in Table 1.

<table>
<thead>
<tr>
<th>EREG1 = nnmx Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>where:</td>
</tr>
<tr>
<td>nn = 00           if no error was found</td>
</tr>
<tr>
<td>= 01,..,08H       if an error was found, nn is the error code</td>
</tr>
<tr>
<td>and: mx = 0xH</td>
</tr>
<tr>
<td>for Test =</td>
</tr>
<tr>
<td>SYS0x; 1 ≤ x ≤ 03H</td>
</tr>
<tr>
<td>ALU0x; 1 ≤ x ≤ 05H</td>
</tr>
<tr>
<td>MEM0x; 1 ≤ x ≤ 00H</td>
</tr>
<tr>
<td>D96A96; x = 0</td>
</tr>
<tr>
<td>DSTISR; x = 0</td>
</tr>
<tr>
<td>DSTHSI; x = 1</td>
</tr>
<tr>
<td>DSTHSO; x = 0</td>
</tr>
<tr>
<td>DSTDQ; x = 0</td>
</tr>
<tr>
<td>DSTHIO; x = 0</td>
</tr>
<tr>
<td>DSTTOV; 0 ≤ x ≤ 1</td>
</tr>
<tr>
<td>DSTXXI; x = 0</td>
</tr>
<tr>
<td>DSTEI; x = 0</td>
</tr>
<tr>
<td>DSTSER; x = 0</td>
</tr>
<tr>
<td>DSTA2D; x = 0</td>
</tr>
<tr>
<td>DSTDST; 0 ≤ x ≤ 1</td>
</tr>
<tr>
<td>DST9FST; x = 0</td>
</tr>
<tr>
<td>DST9F96; x = 0</td>
</tr>
</tbody>
</table>
2.2 Using the General Diagnostics

The General Diagnostics provide a large set of system, ALU and memory tests that can be used in any combination, independent of system configuration or external circuitry. In addition to allowing for a wide flexibility in how a user’s system is externally configured, the tests place minimal requirements on memory maps and interrupt environment.

Except where noted, all tests are interruptible, and maintain Program Status Word and Interrupt Mask integrity. The tests conform to PLM96 conventions, and require only runtime parameters to be passed for such specifics as memory test bounds and ALU test duration. To obtain access to the general diagnostics, the user should declare the needed module names EXTERNAL code segment symbols, and link to:

DIAG96.LIB

The tests are invoked in assembly language by placing the proper parameters on the STACK and CALLing the procedure. In PLM, the tests are run after a function reference is made with the appropriate parameters. The following is an example of an ASM96 call to a memory test:

```
PUSH #4000h
PUSH #5000h
CALL MEM06
CMPB EREG1 + 1,0
BNE Error__Found
```

The diagnostic module called performs a complementary address test on the byte locations between 4000H and 5000H inclusive. If an error is found, the value returned in the word EREG1 will have a non-zero value as its high byte. Also in the case of an error, the MEM06 memory test will place the address of the error in location EREG2. The program D96A96, shown in Appendix D is a working ASM96 example that calls every General Diagnostic Test.

The same memory test could be called in a PLM96 program as follows:

```
Response = MEM06(4000h,5000h);
IF Error$Codes.Number > 255 THEN CALL Error$Found;
```

Since the diagnostics return two words in the PLMSREG locations ICH and IEH, the function MEM06 would be a PROCEDURE of type LONG. Error$Codes would have to be declared a STRUCTURE AT Response, with the word elements Number and Detail so that the error messages returned by the diagnostic can be stored. Number would contain the EREG1 value returned by the test, and Detail would contain EREG2. Response would have to be DECLARED a double word. The program D96P96, shown in Appendix D is a working PLM96 example that calls every General Diagnostic.

The action taken when an error is detected will depend upon the application. For example, the following Error__Found (or Error$Found) routine would output the error codes to a printer or terminal:

```
Error__Found:       Error$Found:      PROCEDURE;
                    PUSHF                     DISABLE
                    PUSH #Message__Ptr__A
                    CALL Send__String
                    CALL output (.Message$Ptr$A,
                        Error$Codes.Number);
                    PUSH EREG1
                    CALL Send__Hex__Word
                    CALL Send__CR__LF
                    PUSH #Message__Ptr__B
                    CALL Send__String
                    CALL output (.Message$Ptr$B,
                        Error$Codes.Detail);
                    Self: GOTO Self;

(Display continues on next page)
PUSH EREG2
CALL Send_Hex_Word
CALL Send_CR_LF
BR $

Message_Ptr_A:
    DCB 27,'ERROR FOUND. Error Number = '

Message_Ptr_B:
    DCB 22,'Error Detail Code is = '

In the Error_Found routine, it is assumed that the subroutines Send_String, Send_Hex_Word, and Send_CR_LF transmit appropriate ASCII codes given the parameters passed to them. Send_String is sent a pointer to a byte string in memory, the first byte of which is the character count. Send_Hex_Word converts the word put on the STACK into the correct four ASCII code bytes and appends the ASCII code for H. Send_CR_LF outputs the ASCII codes to cause a carriage return, followed by a line feed. The PLM routine output would perform similar operations.
2.3 Using the Dynamic Stability Test

The Dynamic Stability Test consists of a set of 8096 interrupt service routines that are designed to run while a user-supplied background task executes. The routines are located in the object file library DST96.LIB, which is contained in the master library DIAG96.LIB. To obtain access to the test, the user should invoke the batch file DSTRL.BAT with the background task file name and directory parameters. For example type:

```
DSTRL \ SOURCE \ BACK
```

Since the interrupt service routines test 8096 on-chip I/O devices, the part under test must reside in a specified hardware environment. Two such environments are available for use with the Dynamic Stability Test. The test may run in either a single chip mode, or a cross-coupled two chip mode. Figures 2 and 3 show the connections required for each configuration. In the single chip mode, output pins are connected to input pins on the same 8096. In the dual chip mode, output pins of one 8096 are connected to the input pins of the other (and vice versa).

To run the test, the user must supply a background task that CALLs an initialization routine (DSTISR) with the specified parameters. After DSTISR returns, the interrupt service routines will begin running. The background task can then perform any function that conforms to the constraints discussed in Section 4. If the user does not wish to write a special background task, one is provided in the module DSTUSR.

The following is an example CALL and a description of the parameters that must be passed to the initialization module (DSTISR).

```
PUSH <RAM segment1 starting address>
PUSH <RAM segment1 ending address>
PUSH <RAM segment2 starting address>
PUSH <RAM segment2 ending address>
PUSH <random seed>
PUSH <random test length>
PUSH <argument1 for Multiply/Divide Core test>
PUSH <argument2 for Multiply/Divide Core test>
PUSH <bit pattern for memory test>
CALL DSTISR
```

The RAM starting and ending addresses form a memory map for the memory tests that DSTISR runs. The internal RAM is always tested. The random seed is the starting point for ALU tests that execute for as many number pairs as is specified in the random test length parameter. Argument1 and argument2 are the operands for a Multiply/Divide test. The bit pattern parameter is used during a memory test of the internal RAM and the memory segments specified.

Section 4 contains more detailed information on using the Dynamic Stability Test, while the next section lists some general restrictions and assumptions that need to be understood to properly use any MCS-96 Diagnostic Library module.
2.4 Restrictions and Assumptions

Some general restrictions and assumptions need to be understood before any DIAG96.LIB programs can be successfully used.

- Pay close attention to the warnings about STACK location in the test modules you use. If you use any of the specialized internal register tests, make sure that the STACK is located externally. Do not partition a region of memory that contains your STACK in any memory test, unless you first move the STACK to an area you already tested.

- All General Diagnostics assume that the WATCHDOG TIMER is either being RESET by an interrupt service routine created by the user, or that it was never enabled. Only SYS02 ever locks out interrupts for a significant period of time. The amount of time they are locked out depends upon the parameters passed.

- The Dynamic Stability Test takes care of the WATCHDOG TIMER within its interrupt service routines. But, do not write to the WATCHDOG before CALLing the initialization subroutine.

- In any Dynamic Stability application, the user's Main Task should not lock out interrupts for more than a few instructions, as the CPU can get quite loaded down with interrupt requests that are very time dependent.
3.0 GENERAL DIAGNOSTICS

The 24 General Diagnostics included in DIAG96.LIB provide a good set of basic memory and ALU confidence tests that can be easily linked to application programs.

The General Diagnostics allow for a wide flexibility in how a user’s system is configured with respect to memory maps and interrupt environment. Except where noted, all tests are interruptible, and maintain Program Status Word and interrupt mask integrity. The tests conform to PLM96 conventions, and require only run-time parameters to be passed for such specifics as memory test bounds and ALU test duration.

The tests are independent to allow specialized diagnostics to be developed as desired. Use just the quick power-up test (SYS02) to verify operation, or use the module that calls all General Diagnostics (D96A96) and let it run continuously for months. A module that performs the most common set of tests is also provided (D96FST).

The tests provided are of four classes: System Tests (SYSnn), ALU Tests (ALUnn), Memory Tests (MEMnn), and Collected Tests (D96xxx). To use any of the modules, from zero to ten parameters are PUSHed onto $e STACK and the test is CALLed. Results are returned in the two word registers beginning at #1CH. The symbolic names for these locations (EREG1 and EREG2) are made PUBLIC if any DIAG96.LIB module is linked. They also may be referenced in PLM$REG for PLM96 programs.

To obtain access to library modules, the user should declare the needed module names EXTERNAL code segment symbols, and link to:

DIAG96.LIB

The next few pages contain a brief overview of each of the four classes of tests. Then, the actions of each test are described in more detail.

System Tests

SYSnn

Common symbol definitions, storage reservations and two common routines are located in SYS01. A reference to any DIAG96.LIB module will cause SYS01 to be linked. SYS02 is meant to be called immediately after a RESET. It checks the special function register status and stack pointer, program status word and timer functionality. SYS03 is a simple program counter test. It does not test the complete range of the counter, and requires external RAM to execute.

SYS01: Common module  
SYS02: RESET test  
SYS03: Program counter exercise

ALU Tests

ALUnn

Five ALU modules are provided for checking ALU functionality. All report errors with a code in EREG1/EREG2.

Addition and subtraction are exercised in ALU01. A special eight-word add and subtract
MCS®-96 Diagnostics Library

is executed to test each adder bit with all possible combinations of a bit operation with and without carry-in.

Unsigned byte multiplication is verified by ALU02. This module simply executes all possible unsigned byte multiplications. Although not elegant, the test is effective. It takes six seconds.

A general test of the multiplication and division functions can be made with ALU03. The module executes all possible combinations of signed and unsigned, byte and word, two and three operand Multiplies and Divides using a specially selected table of numbers as operands.

ALU04 extends the ALU03 test by generating pseudo-random test pairs. The user program simply specifies a seed value for the random number generator, and the number of pairs to generate.

ALU05 is the core module for multiply/divide tests. Both ALU03 and ALU04 call ALU05. The user can also call ALU05 by passing a pair of test arguments. The module executes all possible combinations of signed and unsigned, byte and word, two and three operand Multiplies and Divides using the arguments passed as operands.

ALU01: Table-driven Addition/Subtraction
ALU02: MULUB (all possible arguments)
ALU03: Table-driven Multiply/Divide
ALU04: Pseudo-random Multiply/Divide
ALU05: Multiply/Divide core module

Memory Tests

MEMnn

The DIAG96.LIB MEMnn modules provide tests for register space, external RAM, and ROM. The algorithms used include: walking and galloping ones; walking and galloping zeros; checkerboard patterns; complementary addressing; and checksum verification.

The register tests are in MEM01-MEM05, and MEM0C. With the exception of MEM04, the register tests save the contents of all internal registers except PLMS$REG on the STACK before testing, and restore the data when done. If a faulty location is found, its address is reported. MEM04 is a utility which returns the number of bits set in a specified operand.

The external RAM tests are located in MEM06-MEM0A, and MEM0D. They all return a two-word code upon completion. The calling program must partition the RAM to be tested before calling an external RAM test.

Table 2. Memory Tests

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Internal Registers</th>
<th>External RAM</th>
<th>ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complementary Address</td>
<td>MEM01</td>
<td>MEM06</td>
<td>MEM6B</td>
</tr>
<tr>
<td>Walking Ones</td>
<td>MEM02</td>
<td>MEM07</td>
<td>MEM0F</td>
</tr>
<tr>
<td>Walking Ones/Zeros</td>
<td>MEM02</td>
<td>MEM09</td>
<td>MEM08</td>
</tr>
<tr>
<td>Galloping Ones</td>
<td>MEM03</td>
<td>MEM0A</td>
<td>MEM07</td>
</tr>
<tr>
<td>Galloping Ones/Zeros</td>
<td>MEM04</td>
<td></td>
<td>MEM0C</td>
</tr>
<tr>
<td>Bit Counter</td>
<td>MEM05</td>
<td>MEM0D</td>
<td>MEM0B</td>
</tr>
<tr>
<td>Checkerboard Pattern</td>
<td>MEM0C</td>
<td>MEM0B</td>
<td>MEM0B</td>
</tr>
<tr>
<td>User Specified Pattern</td>
<td>MEM0B</td>
<td></td>
<td>MEM0B</td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Collected Tests

D96xxx

The D96xxx set of modules collects together all, or several, of the General Diagnostics and performs them according to the parameters passed. D96A96 is an ASM96 module that calls all tests. D96P96 is a PLM96 module that calls all tests. D96FST is an ASM96 module that calls a logical selection of tests.

D96A96: All tests / ASM96
D96P96: All tests / PLM96
D96FST: Selection of tests / ASM96
3.1 System Tests

Common Symbols (SYS01)

Brief Description:

This module contains the global symbol declarations and five utilities used by the General Diagnostics.

Assembly Language Calling Sequence:

CALL Get_Psw
or
CALL Put_Psw
or
CALL Get_Parms
or
CALL Stack_Ram
or
CALL Restore_Ram

Get_Psw Action:

USER_PSW := PSW
EREG1 := 0
EREG2 := 0ffff

Put_Psw Action:

PSW := USER_PSW

Get_Parms Action:

PARM2 := Last Parameter put on the STACK
PARM1 := Next to last parameter put on the STACK
USER_PSW := PSW
EREG1 := 0ffff
EREG2 := 0000h

Get_Parms Action:

PSW := USER_PSW

Stack_Ram Action:

PUSH 1aH;
Ptr := 20H
Do While Ptr<100h;
PUSH [Ptr]+
End While;

Restore_Ram Action:

Ptr := 0feh;
Do While Ptr>1eh;
POP [Ptr];
Ptr:=Ptr-2;
End While;
POP 1aH;

Detailed Description:

A call to any General Diagnostic module will cause SYS01 to be linked. This module contains the definition of 4 words of memory used by every module to report errors and store temporary parameters. The STACK routines are used by the internal register tests to save and restore the data in the registers when called. It also INCLUDES an expanded 8096.INC file to provide the PUBLIC declarations of commonly used symbols for the special function registers and constants such as CR and LF.

Nearly all General Diagnostic modules use the routines in SYS01 to save the PSW when called, restore the PSW when returning control to the calling routine, save parameters from the STACK, and initialize the error registers.
System Power-up (SYS02)

Brief Description:

This test is a quick check of the Program Status Word, TIMER1, IOS0, IOS1 and the Interrupt Pending Register. It is meant to be called just after a RESET.

Assembly Language Calling Sequence:

CALL SYS02

When Test Passes:

EREG1 := 0002h  EREG2 := 0000h

If Test Fails:

EREG1 := 0102h on unexpected IOS0 or IOS1 — EREG2 := IOS0 in low byte  
          IOS1 in high byte  
EREG1 := 0202h if TIMER1 does not change — EREG2 := TIMER1  
EREG1 := 0302h if Zero register failed — EREG2 := PSW at Failure  
EREG1 := 0402h if PUSHF/POPF failed — EREG2 := erroneous value found  
EREG1 := 0502h if Sticky bit failed — EREG2 := 3fffh if bit did not set  
          := 0000h if bit did not clear  
EREG1 := 0602h if Carry Flag failed — EREG2 := xxxh  
EREG1 := 0702h on an overflow flag error — EREG2 := 0002h if flags set wrong  
          := xxxh flags cleared wrong  
EREG1 := 0802h if Int. Pending byte failed — EREG2 := offending Int. Pend. value

Detailed Description:

This module verifies that TIMER1 is changing, then attempts to change the value in the ZERO register. Then, a set of PUSHFs and POPFs is done with test values to verify correct action of these instructions. The carry, sticky and overflow bits in the program status word are then tested. Finally, the Interrupt Pending register bits are tested for their ability to be set and cleared. Any unexpected result is reported.

Any error found having to do with the PUSHF/POPF instructions or the PSW, including Interrupt Pending, will cause interrupts to be disabled before returning to the calling module.
Program Counter (SYS03)

Brief Description:

This test writes code into a user selected partition of RAM and executes the code. Elapsed time and special registers are checked for correctness.

Assembly Language Calling Sequence:

PUSH <start address>
PUSH <end address>
CALL SYS03

When Test Passes: If Test Fails:

EREGL := 0003h EREG1 := 0103h if test code returned early
EREG2 := 0000h EREG2 := Early time

Detailed Description:

This module accepts starting and ending addresses for an external RAM partition, adjusts the boundaries to be double word aligned, and writes three lines of code repeatedly into the partition. The code that is written increments a counter then executes two NOPs every 12 state times. The last byte written into the RAM partition is a RET opcode.

After the RAM partition is adjusted and the code written into the RAM, the test puts a return address on the STACK, stores TIMER1 and CALLs the first byte of the RAM. When the last byte of RAM is executed, program control returns to SYS03. TIMER1 is again stored. The test then compares the elapsed time to the expected elapsed time. The value remaining in the counter is also checked for correctness. Any deviations from expected are reported.

Caution: Since interrupts are locked-out while the code in RAM is executing, partitioning more than 4000h bytes of RAM for this test could cause a WATCHDOG TIMER overflow if the watchdog was started before SYS03 is called.
3.2 ALU Tests

Add/Subtract (ALU01)

Brief Description:
This routine adds then subtracts two carefully selected eight-word variables and verifies the results.

Assembly Language Calling Sequence:

CALL ALU01

When Test Passes: If Test Fails:

EREG1 := 0011h EREG1 := 0111h on an addition error
EREG2 := 0000h := 0211h on a subtraction error

Detailed Description:

Two eight-word operands are added together and the results verified. Then, the operands are subtracted and verified. The operands were chosen to exercise every possible combination of two bits and a carry into each bit of the adder. Correctness of the result and the resultant flags is verified.

The operands are:

\[ \begin{align*}
05555AAAA5555AAAAFFFF0000AAAA5555H \\
+05555AAAAAAA5555FFFF0005555AAAAAH \\
0AAAB5555000000000FFFE0000FFFFFFFH \\
05555AAAAAAA5555FFFF0005555AAAAAH \\
-0AAAA5555AAAA55550000FFFF5555AAAAAH \\
0AAAB5555000000000FFFE0000FFFFFFFH
\end{align*} \]

Some versions of SIM96 do not pass this test.
MULUB (ALU02)

Brief Description:

This module simply tests the MULUB instruction for all possible combinations of byte multipliers and multiplicands.

Assembly Language Calling Sequence:

CALL ALU02

When Test Passes:  
EREG1 := 0012h  
EREG2 := 0000h

If Test Fails:  
EREG1 := 0112h on an error  
EREG2 := multiplier/multiplicand

Detailed Description:

This test executes all possible combinations of operands into the MULUB instruction. Results are verified through a method of addition and subtraction as operands cycle. The status of PSW flags is not verified in this routine.

Multiply/Divide Table (ALU03)

Brief Description:

This module sends a specially constructed table of operands through the general Multiply/Divide Core test (ALU05).

Assembly Language Calling Sequence:

CALL ALU03

When Test Passes:  
EREG1 := 0013h  
EREG2 := 0000h

If Test Fails:  
EREG1 := 0115h on a signed error  
EREG1 := 0215h on an unsigned error  
EREG1 := 0315h on a flag error  
EREG2 := offending argument on error

Detailed Description:

This test sends a table of operands through the Multiply/Divide Core test. The 18 operands were selected to exercise all of the hardware multiply and divide control signals.

The operands are:

<table>
<thead>
<tr>
<th>Arg.1,Arg.2</th>
<th>Arg.1,Arg.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D99H, 0FFFH</td>
<td>0FFFH, 9D99H</td>
</tr>
<tr>
<td>9D99H, 5555H</td>
<td>5555H, 0E266H</td>
</tr>
<tr>
<td>0E266H, 0AAAAH</td>
<td>0AAAAH, 1D99H</td>
</tr>
<tr>
<td>1D99H, 5555H</td>
<td>5555H, 9D99H</td>
</tr>
<tr>
<td>9D99H, 0AAAAH</td>
<td>0AAAAH, 0E266H</td>
</tr>
<tr>
<td>0E266H, 0FFFH</td>
<td>0FFFH, 0063H</td>
</tr>
<tr>
<td>0063H, 0055H</td>
<td>0055H, 0066H</td>
</tr>
<tr>
<td>0066H, 00AAH</td>
<td>00AAH, 0063H</td>
</tr>
<tr>
<td>0063H, 00FFH</td>
<td></td>
</tr>
</tbody>
</table>

Some versions of SIM96 will not pass this test.
Multiply/Divide Random (ALU04)

Brief Description:

This module is a pseudo-random number generator that sends pairs of arguments to the Multiply/Divide Core test (ALU05).

Assembly Language Calling Sequence:

```
PUSH <seed>
PUSH <count>
CALL ALU04
```

When Test Passes:

- EREG1 := 0014h
- EREG2 := 0000h

If Test Fails:

```
EREG1 := 0115h on a signed error
        := 0215h on an unsigned error
        := 0315h on a flag error
EREG2 := offending argument on error
```

Detailed Description:

This module first executes the table driven Multiply/Divide test (ALU03). Then, if passed, pseudo-random argument pairs are generated and fed into the generalized Multiply/Divide Test (ALU05). The parameters passed to ALU04 set the random number seed, and the duration of the test.

There is no restriction on the values passed to the test. However, it must be noted that all possible combinations of signed and unsigned, byte and word, two and three operand Multiply/Divides are done at least twice for each pair of arguments sent to ALU05. Each such test takes from 1 to 5 milliseconds depending upon the arguments. Therefore, if large values for the count parameter are selected, the test will be long. For example, 1000h as a count will take about 12 seconds, depending upon the seed. NOTE: Some versions of SIM96 will not pass this test.

The formula used to generate the number pairs is as follows:

\[ X(n+1) = [(0101h + 0001h) \cdot X(n) + 0001h) \mod \text{0fffh} \]

where \( X(0) = \text{seed} \)
Multiply/Divide Core (ALU05)

Brief Description:

This test performs a Divide/re-Multiply sequence for all possible combinations of two or three operand, signed or unsigned, byte or word operations using the arguments passed to it as operands. The results are verified.

Assembly Language Calling Sequence:

PUSH <argument1>
PUSH <argument2>
CALL ALU05

When Test Passes: If Test Fails:

EREG1 := 0015h EREG1 := 0115h on a signed error
EREG2 := 0000h EREG1 := 0215h on an unsigned error
: = 0315h on a flag error

Detailed Description:

This module takes arguments from a calling program and performs upon them all possible combinations of byte or word, two or three operand, signed or unsigned multiplication and division. Argument2 is used to create the high and low words for a word Divide, and the low byte of Argument1 is used as the divisor in a byte Divide.

The test checks multiplication and division by first dividing one operand by the other, then multiplying the quotient by the divisor and adding the remainder. If the result is the original dividend, the operations were correct. However, the possibility of legitimate division overflows must also be considered.

The test first performs a division and checks flag status for correct indication of overflow conditions. If there has been an overflow, the dividend is right shifted by one, the expected result is updated, and the division is performed over. If a division by zero occurred, just the expected result is corrected and the test is continued.

After a division and overflow check/fixup is complete, a re-multiplication occurs and the result verified. Flag status is also verified. If the results are correct, the original operands are reloaded into the test operand registers and the next Divide/re-Multiply combination is begun.

All Divide/Multiply combinations are performed twice. Once with flags set upon entry, and once with flags clear upon entry.

CALLing ALU03 will run a specially selected table of operands through this test. CALLing ALU04 will run a pseudo-random string of operands through this test.
3.3 Memory Tests

Complementary Address (MEM01)
(for registers)

Brief Description:

This module performs a complementary address test on the registers locations 1ah to 0ffh.

Assembly Language Calling Sequence:

CALL MEM01

When Test Passes: If Test Fails:

EREG1 := 0021h EREG1 := 0121h
EREG2 := 0000h EREG2 := address of the error

Detailed Description:

This module performs a simple address and integrity test on register locations 1ah-0ffh. The algorithm stores the value NOT(ADDRESS) in the location pointed to by ADDRESS for the range, then loops through memory again to verify the contents.

Caution: If the STACK is partially internal, the STACK POINTER must be pointing at least 260 bytes into external RAM at the time MEM01 is called. The STACK cannot be entirely internal. The arithmetic flags in the PSW are undefined after execution of MEM01.
Walking Ones/Zeros (MEM02)  
(for registers)

Brief Description:

This module performs a Walking Ones and Zeros test on the internal registers 1ah-0ffh.

Assembly Language Calling Sequence:

CALL MEM02

When Test Passes:                        If Test Fails:
EREG1 := 0022h                           EREG1 := 0122h
EREG1 := 0000h                           EREG2 := address of the error

Detailed Description:

This module performs a Walking Ones and Zeros test on the internal registers.

The Walking Ones memory test first loads zero in all locations to be tested. Then, ones are placed in the first byte of memory, followed by a verification of all locations. Next, the first location is zeroed and ones are loaded into the second location. All memory is again verified. This process continues until all locations have been loaded with ones.

The Walking Zeros memory test works exactly like Walking Ones, except that a zero is "walked" through memory filled with ones, instead of ones being walked through a memory filled with zeros.

Caution: If the STACK is partially internal, the STACK POINTER must be pointing at least 260 bytes into external RAM at the time MEM02 is called. The STACK cannot be entirely internal. The arithmetic flags in the PSW are undefined after execution of MEM02.
Gallopings Ones/Zeros (MEM03) (for registers)

Brief Description:

This module performs a Galloping Ones and Zeros test on the internal registers 1ah-0ffh.

Assembly Language Calling Sequence:

```
CALL MEM03
```

When Test Passes: If Test Fails:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG1</th>
<th>EREG2</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0023h</td>
<td>0123h</td>
<td>0000h</td>
<td>address of the error</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a Galloping Ones and Zeros test on internal registers.

The Galloping Ones algorithm tests memory by first loading zeros into all locations. Then ones are loaded into the first byte and all memory is verified. The verification is done by alternating reads to the first location and locations through all memory. Next, ones are placed in the second location without altering the first. Verification is again performed by alternating reads to the second location and the rest of memory. This process continues until all locations contain ones.

The Galloping Zeros test is similar to Galloping Ones, except that zeros slowly fill a memory filled with ones. In Galloping Ones, ones slowly fill a memory filled with zeros.

Caution: If the STACK is partially internal, the STACK POINTER must be pointing at least 260 bytes into external RAM at the time MEM03 is called. The STACK cannot be entirely internal. The arithmetic flags in the PSW are undefined after execution of MEM03.

Bits Set (MEM04)

Brief Description:

This module returns the number of bits set in the parameter passed to the routine.

Assembly Language Calling Sequence:

```
PUSH test_value
CALL MEM04
```

When All Bits Zero: When One or More Bits Set:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG1</th>
<th>EREG2</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0024h</td>
<td>0124h</td>
<td>0000h</td>
<td>number of bits set</td>
</tr>
</tbody>
</table>

Detailed Description:

This module returns the number of bits that are set in the low byte of the parameter passed to the test. Any addressing mode may be used to put a value on the STACK, but the parameter on the STACK is treated as an immediate value.
Checkerboard Pattern (MEM05)
(for registers)

Brief Description:

This module performs a Checkerboard Pattern test on the internal registers 1ah-0ffh.

Assembly Language Calling Sequence:

```
CALL MEM05
```

When Test Passes:  
If Test Fails:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG1</th>
<th>EREG2</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>:= 0025h</td>
<td>:= 0125h</td>
<td>:= 0000h</td>
<td>:= address of the error</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a checkerboard test on the internal registers. A checkerboard pattern of ones and zeros is written into the physical rows and columns of the 8096 register space. As the pattern is being written, it is repeatedly verified. After the entire pattern is in place, the memory is verified again, complemented, and re-verified.

Caution: If the STACK is partially internal, the STACK POINTER must be pointing at least 260 bytes into external RAM at the time MEM05 is called. The STACK cannot be entirely internal. The arithmetic flags in the PSW are undefined after execution of MEM05.

Complementary Address (MEM06)

Brief Description:

This module performs a complementary address test on the memory partitioned by user supplied pointers.

Assembly Language Calling Sequence:

```
PUSH <start address>
PUSH <end address>
CALL MEM06
```

When Test Passes:  
If Test Fails:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG1</th>
<th>EREG2</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>:= 0026h</td>
<td>:= 0126h</td>
<td>:= 0000h</td>
<td>:= offending address</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a simple address and integrity test on RAM locations partitioned by the parameters passed. The algorithm stores the value NOT(ADDRESS) in the location pointed to by ADDRESS for the range, then loops through memory again to verify the contents.

Caution: Do not partition RAM that contains valid STACK elements.
Walking Ones (MEM07)

Brief Description:

This module performs a Walking Ones Test on the memory partitioned by the user.

Assembly Language Calling Sequence:

```
PUSH <start address>
PUSH <end address>
call MEM07
```

When Test Passes: If Test Fails:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG2</th>
<th>EREG1</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0027h</td>
<td>0000h</td>
<td>0127h</td>
<td>offending address</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a Walking Ones test on the memory partitioned by the calling program. The Walking Ones memory test first loads zero in all locations to be tested. Then, ones are placed in the first byte of memory, followed by a verification of all locations. Next, the first location is zeroed and ones are loaded into the second location. All memory is again verified. This process continues until all locations have been loaded with ones.

Caution: Do not partition RAM that holds valid elements of the STACK. And, execution time increases non-linearly with memory partition widths.

Galloping Ones (MEM08)

Brief Description:

This module performs a Galloping Ones test on memory partitioned by the calling program.

Assembly Language Calling Sequence:

```
PUSH <start address>
PUSH <end address>
call MEM08
```

When Test Passes: If Test Fails:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG2</th>
<th>EREG1</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0028h</td>
<td>0000h</td>
<td>0128h</td>
<td>offending address</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a Galloping Ones test on memory locations partitioned by the calling program. The Galloping Ones algorithm tests memory by first loading zeros into all locations. Then ones are loaded into the first byte and all memory is verified. The verification is done by alternating reads to the first location and locations through all memory. Next, ones are placed in the second location without altering the first. Verification is again performed by alternating reads to the second location and the rest of memory. This process continues until all locations contain ones.

Caution: Do not partition locations that contain valid elements of the STACK. And, execution time increases non-linearly with memory partition widths.
Walking Ones/Zeros (MEM09)

Brief Description:
This module performs a Walking Ones and Zeros test on the memory locations partitioned by the calling program.

Assembly Language Calling Sequence:

```
PUSH <start address>
PUSH <end address>
CALL MEM09
```

When Test Passes: If Test Fails:

```
EREG1 := 0029h
EREG2 := 0000h
```

```
EREG1 := 0129h
EREG2 := offending address
```

Detailed Description:
This module performs a Walking Ones and Zeros test on the memory partitioned by the calling program.

The Walking Ones memory test first loads zero in all locations to be tested. Then, ones are placed in the first byte of memory, followed by a verification of all locations. Next, the first location is zeroed and ones are loaded into the second location. All memory is again verified. This process continues until all locations have been loaded with ones.

The Walking Zeros memory test works exactly like Walking Ones, except that a zero is "walked" through memory filled with ones, instead of ones being walked through a memory filled with zeros.

Caution: Do not partition RAM that contains valid elements of the STACK. And, execution time increases non-linearly with memory partition widths.
Galloping Ones/Zeros (MEM0A)

Brief Description:

This module performs a Galloping Ones and Zeros test on the memory locations partitioned by the calling program.

Assembly Language Calling Sequence:

```assembly
PUSH <starting address>
PUSH <ending address>
call MEMOA
```

When Test Passes:  

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>002Ah</td>
<td>0000h</td>
</tr>
</tbody>
</table>

If Test Fails:  

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>012Ah</td>
<td>offending address</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a Galloping Ones and Zeros test on memory partitioned by the calling program.

The Galloping Ones algorithm tests memory by first loading zeros into all locations. Then ones are loaded into the first byte and all memory is verified. The verification is done by alternating reads to the first location and locations through all memory. Next, ones are placed in the second location without altering the first. Verification is again performed by alternating reads to the second location and the rest of memory. This process continues until all locations contain ones.

The Galloping Zeros test is similar to Galloping Ones, except that zeros slowly fill a memory filled with ones. In Galloping Ones, ones slowly fill a memory filled with zeros.

Caution: Do not partition RAM that contains valid elements of the STACK. And, execution time increases non-linearly with memory partition widths.

Checksum (MEM0B)

Brief Description:

This module calculates a 16 bit checksum for the memory partition specified by the calling program.

Assembly Language Calling Sequence:

```assembly
PUSH <starting address>
PUSH <ending address>
call MEM0B
```

Test Returns:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>012bh</td>
<td>16-bit checksum</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a 16-bit checksum on the region of memory partitioned by the calling program. RAM or ROM may be partitioned. The module is non-destructive to RAM.
User Pattern (MEM0C)  
(for registers)

Brief Description:

This module performs a Checkerboard Pattern test on the internal registers 1ah-0ffh with a user specified bit pattern.

Assembly Language Calling Sequence:

```
PUSH <desired bit pattern>
CALL MEM0C
```

When Test Passes: If Test Fails:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>002Ch</td>
<td>012Ch</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a checkerboard test on the internal registers with the bit pattern specified by the calling program. The pattern is written into the physical rows and columns of the 8096 register space. As the pattern is being written, it is repeatedly verified. After the entire pattern is in place, the memory is verified again, complemented, and re-verified.

Caution: If the STACK is partially internal, the STACK POINTER must be pointing at least 260 bytes into external RAM at the time MEM0C is called. The STACK cannot be entirely internal. The arithmetic flags in the PSW are undefined after execution of MEM0C.

User Pattern (MEM0D)

Brief Description:

This module performs a Checkerboard Pattern test on a specified region of memory with a specified pattern of bits.

Assembly Language Calling Sequence:

```
PUSH <starting address>
PUSH <ending address>
PUSH <bit pattern>
CALL MEM0D
```

When Test Passes: If Test Fails:

<table>
<thead>
<tr>
<th>EREG1</th>
<th>EREG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>002dh</td>
<td>012dh</td>
</tr>
</tbody>
</table>

Detailed Description:

This module performs a checkerboard test on a region of memory that is specified by the calling program using a bit pattern which is also specified. First, the pattern is written into memory. As the pattern is being written, it is repeatedly verified. After the entire pattern is in place, the memory is verified again, complemented, and re-verified.

Caution: Do not partition RAM that contains valid elements of the STACK.
3.4 Collected Tests Modules

ALL Tests in ASM96 (D96A96)

Brief Description:

This module causes every General Diagnostics test to execute.

Assembly Language Calling Sequence:

PUSH <RAM segment1 starting address>
PUSH <RAM segment1 ending address>
PUSH <RAM segment2 starting address>
PUSH <RAM segment2 ending address>
PUSH <random seed>
PUSH <random test length>
PUSH <top of code address>
PUSH <argument1 for Multiply/Divide Core test>
PUSH <argument2 for Multiply/Divide Core test>
PUSH <bit pattern for memory test>
CALL D96A96

When Tests All Pass: When a Test Fails:

EREG1 := 0030h EREG1 := test module error code
EREG2 := code checksum EREG2 := test module detail code

Detailed Description:

This module calls all General Diagnostics using the parameters passed by the calling program. The parameters needed by the test for proper execution specify two areas of external RAM for memory tests, the ending address of code to be checksummed, the seed and length of the random ALU test, two specific arguments to do the Multiply/Divide Core test, and a bit pattern for memory tests.

Execution speed of this test is highly dependent upon the memory partitions and the length requested for the random ALU test. For example, partitioning 1k and 8k regions of memory, and calling for 1000h random ALU tests, the test takes 3 hours to complete. Testing smaller regions of memory (i.e. 1k and 1k) can reduce test time to a few minutes.

Caution: An external STACK must be used with this test, and it must be in a part of memory outside that partitioned during the CALL.
ALL Tests in PLM96 (D96P96)

Brief Description:

This module causes every General Diagnostics test module to execute.

PLM96 Calling Sequence:

D96P96(RAM segment1 starting address, RAM segment1 ending address, RAM segment2 starting address, RAM segment2 ending address, random seed, random test length, top of code address, argument1 for Multiply/Divide Core test, argument2 for Multiply/Divide Core test, bit pattern for memory tests);

When All Tests Pass:

When a Test Fails:

<table>
<thead>
<tr>
<th>WHEN ALL TESTS PASS</th>
<th>WHEN A TEST FAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLMREG := 00F0h</td>
<td>PLMREG := module error code</td>
</tr>
<tr>
<td>PLMREG + 2 := 16-bit checksum</td>
<td>PLMREG + 2 := module detail code</td>
</tr>
</tbody>
</table>

Detailed Description:

This module calls all General Diagnostics using the parameters passed during invocation. The parameters needed by the test for proper execution specify two areas of external RAM for memory tests, the ending address of code to be checksummed, the seed and length of the random ALU test, two specific arguments to do the Multiply/Divide Core test, and a bit pattern for memory tests.

Execution speed of this test is highly dependent upon the memory partitions and the length requested for the random ALU test. For example, partitioning 1k and 8k regions of memory, and calling for 1000h random ALU tests, the test takes 3 hours to complete. Testing smaller regions of memory (i.e. 1k and 1k) can reduce test time to a few minutes.

In his program, the user will have to DECLARE D96P96 an external procedure of the LONG type, with its parameters declared SLOW. The EREG 1 and EREG2 values reported by library modules are placed in the long-word location at PLMSREG.

The DECLARations in D96P96 show how any one General Diagnostic Module could be called from a PLM96 program. Each needed module needs to be DECLARED an external procedure of the LONG type.

Caution: An external STACK must be used with this test, and it must be in a part of memory outside that partitioned during the CALL.
MCS®-96 Diagnostics Library

Selected Tests in ASM (D96FST)

Brief Description:
This is an ASM module that invokes a selected set of General Diagnostic tests.

Assembly Language Calling Sequence:

PUSH <RAM segment1 starting address>
PUSH <RAM segment1 ending address>
PUSH <RAM segment2 starting address>
PUSH <RAM segment2 ending address>
PUSH <random seed>
PUSH <random test length>
PUSH <top of code address>
PUSH <argument1 for Multiply/Divide Core test>
PUSH <argument2 for Multiply/Divide Core test>
PUSH <bit pattern for memory test>
CALL D96FST

When Tests All Pass:  When a Test Fails:
EREgi := 00E0h  EREG1 := test module error code
EREg2 := code checksum  EREG2 := test module detail code

Detailed Description:

This module calls the Power-up and Program Counter tests then all ALU tests. Then, Complementary Address, Galloping Ones/Zeros and Checkerboard tests are run on the internal registers. Finally, Complementary Address and specified pattern tests are done on external memory and the program is checksummed.

The parameters needed by the test for proper execution specify two areas of external RAM for memory tests, the ending address of code to be checksummed, the seed and length of the random ALU test, two specific arguments to do the Multiply/Divide Core test, and a bit pattern for memory tests.

Execution speed of this test is highly dependent upon the memory partitions and the length requested for the random ALU test. For example, partitioning 1k and 8k regions of memory, and calling for 1000h random ALU tests, the test takes about 20 seconds to complete. Testing smaller regions of memory (i.e. 1k and 1k) can reduce test time further.

Caution: An external STACK must be used with this test, and it must be in a part of memory outside that partitioned during the CALL.
4.0 THE DYNAMIC STABILITY TEST

The Dynamic Stability Test is a set of interrupt service routines designed to run over a user's background task in either one stand alone 8097, or two 8097s that are cross-coupled. In the stand alone mode, the chip's output pins are hooked to its input pins. In the dual chip mode, each controller's output pins are tied to the input pins of the other. The minimum configuration for each mode are shown in Figures 2 and 3. See Figure 11 for the circuit diagram of a board that can be jumpered for either configuration.

What is Dynamic Stability?

A "Dynamic Stability" test was developed to enable testing of the 8097 in an asynchronous environment. In the one chip mode, HSO events are synchronized with the HSI.
event capture logic. However, in the cross-coupled mode, HSO events generated by one chip are captured in the HSI unit of another. As long as separate, non-synchronized clock sources are used for each chip, the HSI line events will occur asynchronously to the chip.

To implement a test that could be either stand alone or co-resident without modification, the creation and verification of I/O events needed to be decoupled. Thus the basic structure of the Dynamic Stability Test takes the form of a set of I/O Producers causing events that I/O Consumers verify. Figure 4 gives a macro view of the Producer/Consumer relationship.

---

**Figure 3. 8097 Strapback Configuration Dual Chip Mode**
What Does the Test Do?

Producer/Consumer exchanges were defined to test nearly all of the 8097 I/O capabilities concurrently. Following initialization, the transactions described are carried out by the set of interrupt service routines that make up the Dynamic Stability Test. The following section describes the test initialization. Then the tests performed are briefly described in the Producer/Consumer framework.

Initialization

To get the ball rolling, the background task must first CALL an initialization routine (DSTISR). This routine clears memory, executes the Selected Tests program (D96FST) from the General Diagnostics, and checks for the presence of an external clock on T2CLK. The serial port is then initialized for internal or external baud rate generation based on the presence of an external clock, and sign on messages are sent over the serial channel.

After initial tests are complete, and just prior to initiation of the interrupt service routines, a pulse is sent out on PORT1.3 that is used to synchronize controllers in the two chip mode. (See Figure 5.) Remember that the objective of the Dynamic Stability Test is to test the controllers asynchronously. Therefore, the synchronization is only done to insure that neither controller starts testing before both are ready to begin.

When a controller is ready to synchronize, it places a 0 on the PORT1.3 pin and looks for a 0 on its PORT1.4 pin. When a 0 is seen, the chip delays 600 microseconds, and then PORT1.3 is set high. The chip then loops until PORT1.4 also goes high. Another delay is inserted, and the tests begin. The worst skew between two controllers that can
occur using this method is 9 state times (2.25 μs in a 12 Mhz system). However, the skew should average between four and five state times. In any case, the parts will be far from synchronized shortly after the tests begin. This is fine, as long as the tests begin together.

In a one chip system, this process appears as a 600 microsecond pulse on PORT1.3. (See Figure 6.) The tests begin 600 microseconds after the rising edge.

When synchronization is complete, the interrupt service routines are initialized, interrupts are enabled, and control is returned to the background task. At this point, the testing really begins.

**Producers and Consumers**

The Producer/Consumer exchanges on the 8097 are executed by the interrupt service routines of the *Dynamime Stability Test*. While some interrupt routines contain an entire Producer or Consumer, some are spread through many routines. Figure 7 shows on a broad level the transactions that occur during test execution. Short descriptions of each Producer and Consumer follow, along with an indication of which interrupt routines contain them.

**Serial Producer • DSTSER** The Serial Producer constantly transmits a table of alphabetic and special characters, and test data which includes the current status of the test and the REAL TIME since reset.

**Serial Consumer • DSTSER** The Serial Consumer monitors the data coming over the serial link to see if all the expected characters are transmitted correctly and in the correct order. Transmission of the test data and the REAL TIME is checked by counting characters between carriage returns.

**Port1 Producer • DSTSWT** The Port1 Producer outputs a series of values on Port1 that are contained in a table constructed to test all possible combinations of input and output of ones and zeros. The test producer executes every 5000h TIMER1 counts via the expiration of Software Timer 1.

**Port1 Consumer • DSTSWT** The Port1 Consumer verifies the patterns appearing on Port1 using a table which contains the expected values. The check executes every 1000h TIMER1 counts via the expiration of Software Timer 2.

**A/D Producer • DSTSWT** The A/D Producer continually starts A/D conversions by loading an HSO command to initiate an A/D. The A/D Producer executes every time Software Timer 0 expires.

**A/D Consumer • DSTA2D** The A/D Consumer verifies the result of conversions initiated by the A/D Producer. It then changes the channel set for conversion and loads an HSO command to cause a Software Timer 0 expiration.

---

**SYNCHRONIZATION PULSE IN THE SINGLE CHIP MODE**

![Single Chip Sync Pulse](image)

**Figure 6. Single Chip Sync Pulse**
External Interrupt Producer • DSTH50 • The External Interrupt Producer causes rising edges on HSO.1, which is tied to EXTINT. This Producer executes every time there has been a falling edge on HSO.1.

External Interrupt Consumer • DSTEX1 • The External Interrupt Consumer responds to rising edges on EXTINT. It resets the WATCHDOG TIMER every execution and tests the Test Status Words every 30 executions to see that all tests are running. This Consumer also loads an HSO command to cause a falling edge on HSO.1

PWM Producer • DSTMV • The PWM Producer executes every time there is a timer overflow. In addition to changing the PWM period, it toggles an LED and checks for unexpected T2CLK overflows. There is no PWM Consumer per se, but the PWM output is tied to HSI.1 which is configured to clock T2CLK. In this way T2CLK counts at a known average rate, and is used by the test in a modulo count fashion to generate a real...
time clock. This module is also expandable to include tests that a user might want to execute only periodically.

**HSO Producer ●DSTHSO●** The High Speed Output Producer executes every time an HSO event on HSO.2 or HSO.3 occurs. Varying pulse widths are created on the pins using predetermined tables of values. The minimum pulse width is 1000H; the maximum is 0C000H TIMER1 counts.

**HSI Consumer ●DSTHSI●** The high speed inputs are monitored by the High Speed Input Consumer. The check executes every time an event occurs on HSI.2 or HSI.3. The HSI Consumer verifies that the proper pulse widths appear on the pins, and that the series of pulse widths is in the right order.

**Interrupt BURST Producer ●DSTSWT,DSTHIO,DSTHSO,DSTHSO●** The previous Producer/Consumer transactions either go between controllers in the dual-chip mode, or stay within the same controller in the single-chip mode. However, there is one Dynamic Stability Test that executes invisibly to a co-controller in the dual-chip mode. This test, the Interrupt BURST Test, causes a flood of interrupts that almost fully load the 8097 with interrupt response requests.

The Interrupt BURST Producer causes a complex chain of events that eventually lead to the updating of the REAL TIME Clock. Since the succession of events involves half of the interrupt service routines, the whole process is described here for understanding.

**The Big Picture** — Each time the REAL TIME Clock is ready to be updated, a BURST of interrupts is setup to occur as close together as possible. Figure 8 shows the sequence of events that occur, their dependency on T2CLK and the commands written into the HSO CAM. If you don't need any more detail, skip “The nitty-gritty”.

**The nitty-gritty** — Every time an the A/D Consumer finishes executing it sets up a Software Timer 0 expiration for TIMER1 = TIMER1 + 2. While T2CLK is between 100h and 600h, the A/D Producer (Software Timer 0) causes a new conversion with an HSO command. If T2CLK is greater than 600h, then an HSO command is loaded to cause a falling edge on HSO.O instead of causing an A/D conversion to start. This begins the BURST sequence.

The falling edge on HSO.O causes an HSO interrupt and an HSI interrupt, since HSO.O is tied to HSI.O. The HSO interrupt loads commands to raise HSO.O at T2CLK = 1900h and start an A/D at T2CLK = 18ffh. The HSI interrupt loads no HSO commands.

When T2CLK = 18ffh an A/D conversion is begun. When T2CLK = 1900h a rising edge occurs on HSO.O causing T2CLK to be reset and HSO.HSI and HSI.O interrupt requests to be made. At approximately the same time an A/D conversion completes and the A/D Done interrupt request is made.

The HSO interrupt service causes no further events. The HSI interrupt service routing loads an HSO command to cause a Software Timer 3 interrupt at T2CLK = 0ffh. The A/D Consumer loads an HSO command to cause a Software Timer 0 interrupt at TIMER1 = TIMER1 + 2. When the A/D Producer executes it loads a command to start an A/D conversion at T2CLK = 100h. And the HSI.0 interrupt service routine updates the REAL TIME Clock (the real output from this whole mess).

The last interrupt that is serviced from this BURST is a Software Timer 3 expiration. This is the BURST Checker. It verifies that all interrupts occurred within a reasonable time window, but causes no further events if all tests passed.

All these activities keep the HSO CAM almost fully loaded. So, to ensure that CAM overwrites never occur, two precautions were taken. First, one CAM slot was allocated to four of the tests that use the HSO unit, and two slots were allocated for shared use by the Interrupt BURST process and the A/D conversion process.
The second precaution was to confirm that either the CAM was not full or the HOLDING REGISTER was empty (depending upon the test) before allowing any write to the CAM.

Figure 9 shows the HSO CAM loading over time, with T2CLK as the timebase. External Interrupt, Port1, HSO.2 and HSO.3 events each are allocated the use of one CAM slot all the time. While T2CLK is below 600h, but above 100h, another CAM slot is used by the A/D Done — Start A/D sequence. When T2CLK goes above 600h, two slots are used by the Interrupt BURST process. The BURST events conclude when T2CLK is reset and climbs to 100h. At 100h, the A/D Done — Start A/D sequence being again.

---

**Figure 8. Interrupt BURST Sequencing**

---

7-37
4.1 How to Use DST96

All program modules that are needed to run the Dynamic Stability Test are contained in the DST96 Library (DST96.LIB). This Library is also a part of DIAG96.LIB. To use the test, one or two 8097s must be configured as previously shown. A background task for the Dynamic Stability interrupt service routines must also be provided and linked to DIAG96.LIB. For those who don't wish to write a background task, one is provided (DSTUSR). But, any code may be written which follows some simple rules.

The Software

The software constraints are relatively minor, but they do create incompatibility with PLM96. All background tasks should be written in ASM96.

Minimally, the background task must load the STACK POINTER, PUSH parameters, CALL DSTISR, and go into a loop. Any other code may come after the CALL to DSTISR, as long as:

• Interrupts are never disabled for more than a few instructions;

• No operations to or from special function registers occur (with the exception of reading TIMER1 or T2CLK), and

Other less grave limitations on the main task are that it:

• Be CSEGed at 2080h;
• Write only to EREG1, EREG2, OSEG registers from 40h to 5Ch, or external RAM, (the OSEG is an RL96 technicality, once DSTISR returns control to the MAIN TASK, locations 40h to 5Ch are not touched by the Tests); other registers can be read, but not written to;

• Communicate to the outside world through Port3 and Port4, (these Ports are untouched by the tests), or memory mapped I/O registers;

To provide the Dynamic Stability Test modules for linkage to your program, modify the batch file DSTRL.BAT to suit your system with respect to memory mapping and invoke the batch file with the appropriate background task filename. For example, type:

DSTRL DSTUSR

The Hardware

The Dynamic Stability Test has been designed to allow flexibility in the way output from the tests is used.

Minimally, no output device (printer, terminal) or function generators need to be attached to the test. If the LED attached to Port 2.7 is not flashing, the test failed. However, no other information may be gained.

To support a greater level of debugging (of the test code initially), the test was designed to output status and error information to one 4800 and one 300 baud device. The baud rates are derived from the function generators if present. Figure 10 shows how both devices can be attached to the test.

With this configuration, the test outputs an initialization message to both devices, then selects just the 4800 baud line for monitoring the Serial Port Producer/Consumer transactions. If an error is detected, the 300 baud line is selected for an error information dump.

A diagram of the circuit used in developing the Dynamic Stability Test appears in Figure 11. It is sufficiently general purpose for use in either the single or double chip modes, with or without printers or terminals attached.

The circuit requires that the 8097 I/O signals be present on an SBE-96 compatible 50 pin connector. The circuit also assumes that the analog voltage reference is provided through the cable. Therefore, if you are using the SBE-96, the jumpers to do this need to be in place (jumper numbers vary with the SBE-96 version).

Figure 12 describes how to jumper the Dynamic Stability Test board for one or two chip tests. Figure 13 shows the SBE-96 50 pin connector pinout. The following sections describe in detail the actions of each interrupt service routine in implementing the Producer/Consumer transactions.
Note: Connect J1 of this board to J3 of an SBE-96. For dual chip operation, also connect J3 of a second SBE-96 to J2 on this board. Use the jumper list in Figure 4-11 for the mode used. The function generators are optional. One, two, or none may be used.
Jumper Connections for Single Chip Mode

<table>
<thead>
<tr>
<th>J1</th>
<th>J1 - J2</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 - 37</td>
<td>34 - 28</td>
</tr>
<tr>
<td>23 - 38</td>
<td>35 - 16</td>
</tr>
<tr>
<td>24 - 39</td>
<td>42 - 31</td>
</tr>
<tr>
<td>25 - 26</td>
<td>34 - 32</td>
</tr>
<tr>
<td>26 - 39</td>
<td>42 - 31</td>
</tr>
<tr>
<td>23 - 38</td>
<td>34 - 28</td>
</tr>
<tr>
<td>45 - 48</td>
<td>46 - 29</td>
</tr>
</tbody>
</table>

Also

<table>
<thead>
<tr>
<th>J2</th>
<th>J2 - J1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 4</td>
<td>8 - 10</td>
</tr>
<tr>
<td>15 - 18</td>
<td>21 - 27</td>
</tr>
<tr>
<td>30 - 30</td>
<td>34 - 32</td>
</tr>
<tr>
<td>11 - 11</td>
<td>12 - 12</td>
</tr>
</tbody>
</table>

Jumper Connections for Dual Chip Mode

<table>
<thead>
<tr>
<th>J1 - J2</th>
<th>J1 - J2</th>
<th>J1 - J2</th>
<th>J1 - J2</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 - 37</td>
<td>33 - 33</td>
<td>14 - 14</td>
<td>34 - 28</td>
</tr>
<tr>
<td>23 - 38</td>
<td>36 - 36</td>
<td>4 - 4</td>
<td>45 - 48</td>
</tr>
<tr>
<td>24 - 39</td>
<td>41 - 41</td>
<td>5 - 5</td>
<td>46 - 29</td>
</tr>
<tr>
<td>25 - 26</td>
<td>44 - 44</td>
<td>7 - 7</td>
<td>35 - 16</td>
</tr>
<tr>
<td>42 - 31</td>
<td>47 - 47</td>
<td>10 - 10</td>
<td>J2</td>
</tr>
<tr>
<td>43 - 32</td>
<td>50 - 50</td>
<td>13 - 13</td>
<td>43 - 32</td>
</tr>
</tbody>
</table>

Also

<table>
<thead>
<tr>
<th>J2</th>
<th>J2 - J1</th>
</tr>
</thead>
<tbody>
<tr>
<td>34 - 28</td>
<td>45 - 48</td>
</tr>
<tr>
<td>46 - 29</td>
<td>35 - 16</td>
</tr>
<tr>
<td>E2 - E5</td>
<td>E1 - E4</td>
</tr>
</tbody>
</table>

Figure 12. Dynamic Stability Board Jumper List

Figure 13. SBE-96 J3 Pinout
4.2 Test Module Descriptions

DST Initialization (DSTISR)

Brief Description:

This module is the invocation and initialization code for the Dynamic Stability Test.

Assembly Language Calling Sequence:

```
PUSH <RAM segment1 starting address>
PUSH <RAM segment1 ending address>
PUSH <RAM segment2 starting address>
PUSH <RAM segment2 ending address>
PUSH <random seed>
PUSH <random test length>
PUSH <top of code address>
PUSH <argument1 for Multiply/Divide Core test>
PUSH <argument2 for Multiply/Divide Core test>
PUSH <bit pattern for memory test>
CALL DSTISR
```

When All Tests Pass:

```
EREG1 := 0040h
EREG2 := 0000h
```

When a Test Fails:

```
EREG1 := 0140h on abnormal RESET
EREG1 := 0240h if T2CLK won’t change
EREG1 := 0340h if T2RST did not work
EREG1 := 0440h if IO0.1 did not work

EREG2 := TIMER1
EREG2 := xxxxh
EREG2 := xxxxh
EREG2 := xxxxh
```

Detailed Description:

This module initializes the registers used by Dynamic Stability Test Modules, checks to see if there is an external clock present, tests T2CLK counting and reset functionality, and outputs initialization messages to the two output devices. The selected tests module (D96FST) from the General Diagnostics is also executed using the parameters specified.

When all initialization tests are passed, then a synchronization is performed to place the two processors in a dual-chip mode test in close sync. The PORT pins are used as to perform the handshaking synchronization. After synchronization, all Dynamic Stability Tests are activated and control is returned to the user program.
MCS®-96 Diagnostics Library

External Interrupts (DSTEXI)

Brief Description:

This module executes every time there is a rising edge on the EXTINT pin. The test resets the WATCHDOG TIMER and verifies execution of all Dynamic Stability routines.

If Test Fails:

EREG1 := 01A0h if a test did not execute
EREG2 := Number of Shifts done

Detailed Description:

This routine executes every time there is a rising edge on the EXTINT pin, causing an external interrupt. Each execution, the WATCHDOG TIMER is reset and an HSO command to clear the HSO.1 pin in 1000h TIMER1 counts is loaded into the CAM. The HSO routine that responds to that event will cause HSO.1 to go high, thus causing another vector to DSTEXI.

Every 30h executions of this module, the Test Status Words are NOTed and then NORMaLized to see if any test did not execute. If any bit in the Test Status Words is left set after being complemented, the NORMAL instruction will leave the most significant bit set, indicating an error. If there was no error, the TSWORDs are cleared. The user can change a mask in DSTEXI to enable checking of any of the currently spare bits in TSWORD. The TSWORD bit map is as follows:

<table>
<thead>
<tr>
<th>EREG2 BIT #</th>
<th>TSWORD2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 0Ah 0Bh 0Ch 0Dh 0Eh 0Fh</td>
</tr>
<tr>
<td></td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>SPARE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EREG2 BIT #</th>
<th>TSWORD1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0Fh 0Eh 0Dh 0Ch 0Bh 0Ah 09 08 07 06 05 04 03 02 01 0</td>
</tr>
<tr>
<td></td>
<td>HSO.3 HSO.2 HSO.1 HSO.0 HSI.3 HSI.2 HSI.0 TIMERS</td>
</tr>
<tr>
<td></td>
<td>SOFTWARE TIMER 3 INTERRUPT BURST CONSUMER</td>
</tr>
<tr>
<td></td>
<td>SOFTWARE TIMER 2 PORT1 CONSUMER</td>
</tr>
<tr>
<td></td>
<td>SOFTWARE TIMER 1 PORT1 PRODUCER</td>
</tr>
<tr>
<td></td>
<td>SOFTWARE TIMER 0 A/D PRODUCER/BURST PRODUCER</td>
</tr>
<tr>
<td></td>
<td>A/D CONSUMER</td>
</tr>
<tr>
<td></td>
<td>SERIAL PORT CONSUMER</td>
</tr>
<tr>
<td></td>
<td>SERIAL PORT PRODUCER</td>
</tr>
<tr>
<td></td>
<td>REAL TIME CLOCK</td>
</tr>
</tbody>
</table>

Figure 14. Test Status Word Bit Map
Serial Port (DSTSER)

Brief Description:

This module contains the Serial Port Consumer and Producer routines for the **Dynamic Stability Tests**. It is executed on every Serial Interrupt.

If Test Fails:

EREG1 := 01B0h if a bad character was received
EREG2 := actual received character
EREG1 := 02B0h if an incorrect number of characters came between carriage returns
EREG2 := actual count

Detailed Description:

This interrupt service routine executes every time there is a Serial Interrupt. The data that is transmitted and checked by the test consists of first, the alphabet and some special characters; second, the current REAL TIME; and finally, the bit representation of the Test Status Words. The receiver verifies the alphabet and funny characters and counts characters until a carriage return. The following is an example of what the output looks like.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ*%&][001:23:59.61 111111011101111111001111
```

The code first checks for a Receive Done flag. If a receive just completed, the receive buffer is emptied and checked for validity. If the received character is a carriage return, then the count since the last carriage return is checked for correctness.

After the receive service has finished, or if there was no receive, DSTSER then checks for the Transmit Done flag. If more transmits can be made, the next data byte is loaded into the transmit buffer. If the data is exhausted, a carriage return is sent, and the routine is set to transmit the first data byte again.
Software Timers (DSTSWT)

Brief Description:

This module is executed every time a Software Timer Interrupt expires. The routine includes the Port1 Producer and Consumer, the A/D producer, and the Interrupt Burst control and verification code.

If a Test Fails:

EREG1 := 01D0h If an unexpected value is found on Port 1
EREG2 := expected value in high byte, actual value in low byte
EREG1 := 02D0h A/D Done interrupt did not occur within BURST window
EREG2 := Time between A/D done and Software Timer 0
EREG1 := 03D0H REAL TIME update did not occur within BURST window
EREG2 := Time between REAL TIME update and Software Timer 0
EREG1 := 04D0H HSO.0 response did not occur within BURST window
EREG2 := Time between HSO.0 interrupt and Software Timer 0
EREG1 := 05D0H HSI(.0) response did not occur within BURST window
EREG2 := Time between HSI(.0) service and Software Timer 0
EREG1 := 01D1H Invalid T2CLK value reached
EREG2 := T2CLK found
EREG1 := 02D1H Test reached an illegal Software Timer 0 state
EREG2 := the illegal case jump that was made

Detailed Description:

This module is called every time a Software Timer expires and causes and interrupt. Software timers are used by the A/D Done — A/D Trigger Sequence, the Interrupt Burst Sequence, and the Port1 Producer and Port1 Consumer.

When Software Timer 0 expires, a case jump is done on the BURST_STATE variable to sequence the A/D and interrupt BURST process to the appropriate state. Depending upon the value of T2CLK and the state of the A/D converter, either an A/D conversion is initiated or HSO.0 is set to go low to begin the interrupt BURST events.

When Software Timer1 expires, a new value is written to Port1 from a table constructed to test all combinations of input/output states on the quasi-bidirectional port pins. The HSO CAM is also loaded with a command to cause Software Timer1 to overflow again in 5000h TIMER1 counts.

When Software Timer 2 expires, Port1 is read and compared to a table of expected entries. If the value is correct, then an HSO command is loaded into the CAM to cause another Software Timer 2 expiration in 1000h TIMER1 counts. If the value is not correct, the next entry in the Table is checked. If there is still no match, an error is reported. If there is a match, the CAM loading occurs and Software Timer 3 is checked for expiration.

If Software Timer 3 has expired, then the flurry of BURST interrupts should have just occurred. The routine checks to see that each event happened within a reasonable time window. If the checks pass, then the routine exists with no further action.
Real Time Clock (HSI0) (DSTH10)

Brief Description:

This routine executes every time there is a rising edge on HSI.0 and updates the real time clock value.

When Module Executes:

REAL__TIME := REAL__TIME + .204 seconds

Detailed Description:

This module is the HSI.0 interrupt service routine. On each rising edge of HSI.0, the value in the REAL TIME clock buffer is updated to reflect the passing of 1900h T2CLK counts. Since the PWM output is tied to T2CLK, and the average time between edges is 31.875 μs in a 12 MHz system, then 1900h T2CLK counts represents .204 seconds.

Execution of this module occurs during the interrupt BURST events. No action other than updating the REAL TIME clock is taken in this routine.

High Speed Outputs (DSTHSO)

Brief Description:

This module manages the pulse width outputs on HSO.2 and HSO.3, and causes the Manager test to execute.

Detailed Description:

Every time an HSO command is executed that has the Interrupt bit set, this program executes. The routine manages the pulse widths on HSO lines two and three, and causes the Manager module to execute at the right time.

When a falling edge has been caused on either HSO.2 or HSO.3, DSTHSO loads a command into the CAM to cause a rising edge on the same line at a time that gives the line a low pulse width equal to a predetermined table value. Rising edges cause analogous responses. The tables used cause low and high pulse widths that vary from 1000h and 0C000h. The length of the tables differ by one so that all combinations of low and high table times occur.

When a falling edge was caused on HSO.1, the routine loads a command into the CAM to cause a rising edge on the same line two TIMER1 counts later. Since HSO.1 is tied to the EXTINT pin, rising edges cause the Manager Routine to execute.
High Speed Inputs (DSTHSI)

Brief Description:
This module does the verification of events on the HSI lines and initiates some interrupt BURST events when appropriate.

If a Test Fails:

- EREG1 := 0161h if a high pulse on HSI.2 had an unexpected width
- EREG2 := difference between actual and expected pulse width

- EREG1 := 0261h if a low pulse on HSI.2 had an unexpected width
- EREG2 := difference between actual and expected pulse width

- EREG1 := 0361h if a high pulse on HSI.3 had an unexpected width
- EREG2 := difference between actual and expected pulse width

- EREG1 := 0461h if a low pulse on HSI.3 had an unexpected width
- EREG2 := difference between actual and expected pulse width

- EREG1 := 0561h if the HSI unit indicated that an HSI.1 event occurred
- EREG2 := the time recorded in the FIFO

Detailed Description:
This module executes every time an event is loaded into the HSI Holding Register. Verification of pulse widths on HSI.2 and HSI.3 is done from tables of expected values. Any deviation is reported as an error.

If the test detects a negative transition on HSI.0, then commands are loaded into the HSO CAM to start an A/D at T2CLK = 18fh and to set HSO.0 high at T2CLK = 1900h. This results in an HSO, HSI, HSI.0 and A/D Done interrupt requests to occur at approximately the same time — approaching a full demand on interrupt service.

When a rising edge on HSI.0 is detected, an HSO command is loaded into the CAM to cause a Software Timer 3 interrupt when T2CLK = 100h. The Software Timer 3 interrupt service will check to see that all burst events happened fast enough.

HSI.1 events are disabled from the FIFO. Any event detected on this line is reported as an error.
A/D Conversion Complete (DSTA2D)

Brief Description:
This module executes every time an A/D conversion is complete. The conversion result is checked for correctness, the A/D converter is setup to convert on the next channel when initiated by an HSO command, and an HSO command to cause a Software Timer 0 expiration is loaded.

If Test Fails:
EREG1 := 01C0h on a conversion error
EREG2 := channel on which error occurred

Detailed Description:
This module executes every time an A/D conversion is complete. The conversion result is checked against a test table for correctness, and the A/D converter is setup to convert on the next channel when initiated by an HSO command. An HSO command to cause a Software Timer 0 expiration in 0002h TIMER1 counts is loaded just prior to exiting the module.

While T2CLK has a value between 100h and 600h, A/D conversions are initiated by the Software Timer 0 Interrupt service routine. When T2CLK goes above 600h, an A/D conversion is initiated by the HSO.O interrupt service routine.

Given the possibility of additive error in 5% resistors, the conversion is tested to only six bits of accuracy.

Timer Overflows (DSTTOV)

Brief Description:
This module toggles a port pin tied to an LED, manages the PWM output, performs some simple tests, and is expandable to allow inclusion of user written tests.

If Test Fails:
EREG1 := 0190h if T2CLK had an overflow indication
EREG2 := T2CLK a the time the error was found

Detailed Description:
This module executes every time TIMER1 or T2CLK overflow. Only TIMER1 overflows are valid however, so T2CLK overflows are flagged as an error. Each overflow, a new period is loaded into the PWMCONTROL register from a table of pulse periods. If an LED is connected, it will appear to slowly change in intensity. Port2.7 is also toggled in this routine to light another LED.

This interrupt routine can be expanded with special tests that are to execute on a periodic basis. Any of the spare bits in the Test Status Words can also be used by specialized tests. They will be checked by the External Interrupt service routine with a simple change in a bit mask.
Macro Module (DSTMAC)

Brief Description:

This module contains four macros used by the Dynamic Stability Test.

Assembly Language Invocation:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSTATUS</td>
<td>Temp_Register</td>
</tr>
<tr>
<td>or</td>
<td>(RI, TI)</td>
</tr>
<tr>
<td>SPWAIT</td>
<td>Label</td>
</tr>
<tr>
<td>or</td>
<td>RESET_WATCHDOG</td>
</tr>
</tbody>
</table>

Detailed Description:

The SPSTATUS Macro is used to ORB the Serial Port Status Register to a temp register. The Macro needs to be used to work around a bug in the 809x-90.

The SPWAIT Macro is used to cause program execution to halt and wait for an RI or TI flag, depending upon which is specified.

The BR_ON_ERROR Macro tests the high byte of EREG1 and jumps to the label if the byte is not zero. This can be used every time a General Diagnostic completes since the detection of any error will cause the high byte of EREG1 to be non-zero.

The RESET_WATCHDOG Macro does just what it says. The WATCHDOG TIMER is reset by writing the correct sequence to location 0Ah.

To access a DSTMAC macro, this module must be $INCLUDEd.
Error Procedure (DSTERR)

Brief Description:

This module is called if any error is detected in the Dynamic Stability Test. Information about the error is output over the serial port, and the test is restarted.

Assembly Language Calling Sequence:

CALL Error_Proc

Detailed Description:

This module is called on detection of any error in the Dynamic Stability Test. When called, the procedure:

• disables interrupts,
• saves any rapidly changing values (TIMER1, T2CLK, HSO_STATUS, ...),
• waits for a serial transmit in progress to complete,
• waits for the current serial receive to complete,
• empties eight entries from the HSI_FIFO,
• transmits an open loop sync sequence in case a co-controller is stuck in the sync routine, and
• waits a few hundred milliseconds to ensure that a co-controller has also detected a failure.

After these steps have been taken, the DSTERR de-selects the 4800 baud line, selects the 300 baud line, and outputs error messages. These messages include the Error Code (EREG1), the Detail Code (EREG2), the address of the line in the test which found the error, and the REAL TIME since reset.

Following the error messages, the procedure dumps the data contained in the registers and the external error buffer out over the serial port to the 300 baud device.

Finally, a RST instruction followed by a branch to the RST instruction is executed. If the WATCHDOG TIMER is externally disabled, the test will stay in this loop. If the WATCHDOG TIMER is not disabled, the test chip will reset, and the Dynamic Stability Test will reinitialize.

DST Example User Code (DSTUSR)

Brief Description:

This is an example program that initiates the Dynamic Stability Test and then executes some General Diagnostics as a background task.

Detailed Description:

DSTUSR sends parameters defined at assembly time to the DST initialization routine (DSTISR). When control returns to DSTUSR, the example repeatedly executes ALU01, ALU02, ALU04, ALU05 and MEM0A. It takes two minutes (with the given memory parameters) for the DSTUSR background task to cycle once while interrupts are running.

When creating a custom background task, using this example program as a template will speed development.
APPENDICES

APPENDIX A • DIAG96.LIB Error Messages by EREG1 Code
APPENDIX B • DIAG96.LIB Error Messages by Module Name
APPENDIX C • Description of DIAG96.LIB Batch Files
APPENDIX D • Example Program Listings
  — D96A96
  — D96P96
  — D96FST
  — DSTUSR
APPENDIX A

DIAG96.LIB Error Messages by EREG1 Code

0000  No Message
       EREG2 = 0fffh
       MODULE = SYS01/Common Symbols

0002  All Tests Passed
       EREG2 = 0000
       MODULE = SYS02/System Power-up

0003  All Tests Passed
       EREG2 = 0000
       MODULE = SYS03/Program Counter

0011  All Tests Passed
       EREG2 = 0000
       MODULE = ALU01/Add/Subtract

0012  All Tests Passed
       EREG2 = 0000
       MODULE = ALU02/MULUB

0013  All Tests Passed
       EREG2 = 0000
       MODULE = ALU03/Multiply/Divide Table

0014  All Tests Passed
       EREG2 = 0000
       MODULE = ALU04/Multiply/Divide Random

0015  All Tests Passed
       EREG2 = 0000
       MODULE = ALU05/Multiply/Divide Core

0021  All Tests Passed
       EREG2 = 0000
       MODULE = MEM01/Complementary Address (Registers)

0022  All Tests Passed
       EREG2 = 0000
       MODULE = MEM02/Walking Ones/Zeros (Registers)

0023  All Tests Passed
       EREG2 = 0000
       MODULE = MEM03/Galloping Ones/Zeros (Registers)

0024  No bits were set in the byte tested
       EREG2 = 0000
       MODULE = MEM04/Bits Set

0025  All Tests Passed
       EREG2 = 0000
       MODULE = MEM05/Checkerboard Pattern (Registers)

0026  All Tests Passed
       EREG2 = 0000
       MODULE = MEM06/Complementary Address
0027 All Tests Passed
EREG2 = 0000
MODULE = MEM07/Walking Ones

0028 All Tests Passed
EREG2 = 0000
MODULE = MEM08/Galloping Ones

0029 All Tests Passed
EREG2 = 0000
MODULE = MEM09/Walking Ones/Zeros

002A All Tests Passed
EREG2 = 0000
MODULE = MEM0A/Galloping Ones/Zeros

002C All Tests Passed
EREG2 = 0000
MODULE = MEM0C/User Pattern (Registers)

002D All Tests Passed
EREG2 = 0000
MODULE = MEM0D/User Pattern

0030 All Tests Passed, checksum is ready
EREG2 = 16-bit checksum
MODULE = D96A96/ALL Tests in ASM96

0040 Initialization completed satisfactorily
EREG2 = 0000
MODULE = DSTISR/DST Initialization

00E0 All Tests Passed, checksum is over range specified
EREG = 16-bit checksum
MODULE = D96FST/Selected Tests in ASM

00F0 All Tests Passed, checksum is ready
EREG2 = 16-bit checksum
MODULE = D96P96/ALL Tests in PLM96

0102 I/O Status Registers were unexpected
EREG2 = IOS0 in low byte, IOS1 in high byte
MODULE = SYS02/System Power-up

0103 Test Code Returned Early
EREG2 = Early Time
MODULE = SYS03/Program Counter

0111 An Addition error occurred
EREG2 = offending argument when the error occurred
MODULE = ALU01/Add/Subtract

0112 Incorrect multiplication result was detected
EREG2 = Multiplier/Multiplicand
MODULE = ALU02/MULUB

0115 A signed operation failed
EREG2 = offending argument on error
MODULE = ALU03/Multiply/Divide Table
0115 A signed operation failed
EREG2 = offending argument on error
MODULE = ALU04/Multiply/Divide Random

0115 A signed operation failed
EREG2 = offending argument on error
MODULE = ALU05/Multiply/Divide Core

0121 A memory location failed
EREG2 = address of the error
MODULE = MEM01/Complementary Address (Registers)

0122 A memory location failed
EREG2 = address of error
MODULE = MEM02/Walking Ones/Zeros (Registers)

0123 A memory location failed
EREG2 = address of the error
MODULE = MEM03/Galloping Ones/Zeros (Registers)

0124 At least one bit was set in the byte tested
EREG2 = number of bits set
MODULE = MEM04/Bits Set

0125 A memory location failed
EREG2 = address of the error
MODULE = MEM05/Checkerboard Pattern (Registers)

0126 A memory location failed
EREG2 = address of error
MODULE = MEM06/Complementary Address

0127 A memory location failed
EREG2 = address of the error
MODULE = MEM07/Walking Ones

0128 A memory location failed
EREG2 = address of the error
MODULE = MEM08/Galloping Ones

0129 A memory location failed
EREG2 = address of the error
MODULE = MEM09/Walking Ones/Zero

012A A memory location failed
EREG2 = address of the error
MODULE = MEM0A/Galloping Ones/Zeros

012B 16-bit Checksum is ready
EREG2 = 16-bit Checksum
MODULE = MEM0B/Checksum

012C A memory location failed
EREG2 = address of the error
MODULE = MEM0C/User Pattern (Registers)

012D A memory location failed
EREG2 = address of the error
MODULE = MEM0D/User Pattern
0140 An abnormal RESET occurred
EREG2 = TIMER1
MODULE = DSTISR/DST Initialization

0161 A high pulse on HSI.2 had an unexpected width
EREG2 = difference between actual and expected pulse width
MODULE = DSTHSI/High Speed Inputs

0190 An overflow of T2CLK was indicated
EREG2 = TIMER1
MODULE = DSTTOV/Timer Overflows

01A0 One or more DST Module did not execute on time
EREG2 = Number of SHIFTS done
MODULE = DSTEXI/External Interrupt (Supervisor)

01B0 An unexpected serial character was received
EREG2 = Bad character received
MODULE = DSTSERI/Serial Port

01C0 An unexpected A/D conversion result was found
EREG2 = Channel number of unexpected result
MODULE = DSTA2D/A/D Conversion Complete

01D0 Found unexpected value on PORT1
EREG2 = expected value in high byte, actual in low byte
MODULE = DSTSWT/Software Timers

01D1 Invalid T2CLK value reached
EREG2 = T2CLK
MODULE = DSTSWT/Software Timers

0202 TIMER1 did not change over time
EREG2 = TIMER1
MODULE = SYS02/System Power-up

0203 Test Code Returned Late
EREG2 = Late Time
MODULE = SYS03/Program Counter

0211 A Subtraction error occurred
EREG2 = offending argument when the error occurred
MODULE = ALU01/Add/Subtract

0215 An unsigned operation failed
EREG2 = offending argument on error
MODULE = ALU03/Multiply/Divide Table

0215 An unsigned operation failed
EREG2 = offending argument on error
MODULE = ALU04/Multiply/Divide Random

0215 An unsigned operation failed
EREG2 = offending argument on error
MODULE = ALU05/Multiply/Divide Core

0240 T2CLK will not change
EREG2 = xxxx
MODULE = DSTISR/DST Initialization
0261 A low pulse on HSI.2 had an unexpected width
EREG2 = difference between actual and expected pulse width
MODULE = DSTHSI/High Speed Inputs

02B0 A carriage return was received out of sequence
EREG2 = number of characters since a carriage return
MODULE = DSTSER/Serial Port

02D0 A/D Done did not occur within BURST window
EREG2 = Time between A/D done and Software Timer 0
MODULE = DSTSWT/Software Timers

02D1 Test reached an illegal Software Timer 0 state
EREG2 = Illegal case jump made
MODULE = DSTSWT/Software Timers

0302 Zero Register was found to change
EREG2 = Program Status Word At Failure
MODULE = SYS02/System Power-up

0303 Counter Register contained unexpected value
EREG2 = Erroneous Counter Value
MODULE = SYS03/Program Counter

0311 A flag error occurred
EREG2 = offending argument when the error occurred
MODULE = ALU01/Add/Subtract

0315 A flag error occurred
EREG2 = offending argument on error
MODULE = ALU03/Multiply/Divide Table

0315 A flag error occurred
EREG2 = offending argument on error
MODULE = ALU04/Multiply/Divide Random

0315 A flag error occurred
EREG2 = offending argument on error
MODULE = ALU05/Multiply/Divide Core

0340 T2RST pin would not RESET T2CLK
EREG2 = xxxx
MODULE = DSTISR/DST Initialization

0361 A high pulse on HSI.3 had an unexpected width
EREG2 = difference between actual and expected pulse width
MODULE = DSTHSI/High Speed Inputs

0391 Illegal Opcode

03D0 REAL TIME update did not occur within BURST window
EREG2 = Time between REAL TIME update and Software Timer 0
MODULE = DSTSWT/Software Timers

0402 PUSHF or POPF failed
EREG2 = Erroneous PUSHed or POPed value found
MODULE = SYS02/System Power-up
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0440 | 10C0.1 would not RESET T2CLK  
EREG2 = xxxx  
MODULE = DSTISR/DST Initialization |
| 0461 | A low pulse on HSI.3 had an unexpected width  
EREG2 = difference between actual and expected pulse width  
MODULE = DSTHSI/High Speed Inputs |
| 04D0 | HSO.0 response did not occur within BURST window  
EREG2 = Time between HSO.0 update and Software Timer 0  
MODULE = DSTSWT/Software Timers |
| 0502 | Sticky Bit would not set  
EREG2 = 3fffh  
MODULE = SYS02/System Power-up |
| 0502 | Sticky Bit would not clear  
EREG2 = 0000  
MODULE = SYS02/System Power-up |
| 0561 | HSI unit indicated an HSI.1 event occurred  
EREG2 = Time recorded in HSI FIFO  
MODULE = DSTHSI/High Speed Inputs |
| 05D0 | HSI(.0) response did not occur within BURST window  
EREG2 = Time between HSI(.0) service and Software Timer 0  
MODULE = DSTSWT/Software Timers |
| 0602 | Carry Flag Test Failed  
EREG2 = xxxx  
MODULE = SYS02/System Power-up |
| 0702 | Overflow flags would not set correctly  
EREG2 = 0002h  
MODULE = SYS02/System Power-up |
| 0702 | Overflow flags would not clear correctly  
EREG2 = xxxx  
MODULE = SYS02/System Power-up |
| 0802 | Interrupt Pending Register failed read/write test  
EREG2 = offending Interrupt Pending byte  
MODULE = SYS02/System Power-up |
| xx91 | (user defined)  
EREG2 = (user defined)  
MODULE = DSTTOV/Timer Overflows |
APPENDIX B

DIAG96.LIB Error Messages by Module Name

ALU01 Add/Subtract
   0011 All Tests Passed
   EREG2 = 0000
   0111 An Addition error occurred
   EREG2 = offending argument when the error occurred
   0211 A Subtraction error occurred
   EREG2 = offending argument when the error occurred
   0311 A flag error occurred
   EREG2 = offending argument when the error occurred

ALU02 MULUB
   0012 All Tests Passed
   EREG2 = 0000
   0112 Incorrect multiplication result was detected
   EREG2 = Multiplier/Multiplicand

ALU03 Multiply/Divide Table
   0013 All Tests Passed
   EREG2 = 0000
   0115 A signed operation failed
   EREG2 = offending argument on error
   0215 An unsigned operation failed
   EREG2 = offending argument on error
   0315 A flag error occurred
   EREG2 = offending argument on error

ALU04 Multiply/Divide Random
   0014 All Tests Passed
   EREG2 = 0000
   0115 A signed operation failed
   EREG2 = offending argument on error
   0215 An unsigned operation failed
   EREG2 = offending argument on error
   0315 A flag error occurred
   EREG2 = offending argument on error

ALU05 Multiply/Divide Core
   0015 All Tests Passed
   EREG2 = 0000
   0115 A signed operation failed
   EREG2 = offending argument on error
   0215 An unsigned operation failed
   EREG2 = offending argument on error
   0315 A flag error occurred
   EREG2 = offending argument on error
MCS®-96 Diagnostics Library

D96A96 All Tests in ASM96
0030 All Tests Passed, checksum is ready
EREG2 = 16-bit checksum

D96FST Selected Tests in ASM
00E0 All Tests Passed, checksum is over range specified
EREG2 = 16-bit checksum

D96P96 ALL Tests in PLM96
00F0 All Tests Passed, checksum is ready
EREG2 = 16-bit checksum

DSTA2D A/D Conversion Complete
01C0 An unexpected A/D conversion result was found
EREG2 = Channel number of unexpected result

DSTEX1 External Interrupt (Supervisor)
01A0 One or more DST Module did not execute on time
EREG2 = Number of SHIFTs done

DSTHSI High Speed Inputs
0161 A high pulse on HSI.2 had an unexpected width
EREG2 = difference between actual and expected pulse width

0261 A low pulse on HSI.2 had an unexpected width
EREG2 = difference between actual and expected pulse width

0361 A high pulse on HSI.3 had an unexpected width
EREG2 = difference between actual and expected pulse width

0461 A low pulse on HSI.3 had an unexpected width
EREG2 = difference between actual and expected pulse width

0561 HSI unit indicated an HSI.1 event occurred
EREG2 = Time recorded in HSI FIFO

DSTISR DST Initialization
0040 Initialization completed satisfactorily
EREG2 = 0000

0140 An abnormal RESET occurred
EREG2 = TIMER1

0240 T2CLK will not change
EREG2 = xxxx

0340 T2RST pin would not RESET T2CLK
EREG2 = xxxx

0440 IOCO.1 would not RESET T2CLK
EREG2 = xxxx

DSTSER Serial Port
01B0 An unexpected serial character was received
EREG2 = Bad character received

02B0 A carriage return was received out of sequence
EREG2 = number of characters since a carriage return
DSTS 의원es oun-96 Dianonics Library

DSTSWT  Software Timers
  01D0 Found unexpected value on PORT1
      EREG2 = expected value in high byte, actual in low byte

  01D1 Invalid T2CLK value reached
      EREG2 = T2CLK

  02D0 A/D Done did not occur within BURST window
      EREG2 = Time between A/D done and Software Timer 0

  02D1 Test reached an illegal Software Timer 0 state
      EREG2 = Illegal case jump made

  03D0 REAL TIME update did not occur within BURST window
      EREG2 = Time between REAL TIME update and Software Timer 0

  04D0 HSO.0 response did not occur within BURST window
      EREG2 = Time between HSO.0 update and Software Timer 0

  05D0 HSI(.0) response did not occur within BURST window
      EREG2 = Time between HSI(.0) service and Software Timer 0

DSTTOV  Timer Overflows
  0190 An overflow of T2CLK was indicated
      EREG2 = TIMER1

  xx91 (user defined)
      EREG2 = (user defined)

MEM01  Complementary Address (Registers)
  0021 All Tests Passed
      EREG2 = 0000

  0121 A memory location failed
      EREG2 = address of the error

MEM02  Walking Ones/Zeros (Registers)
  0022 All Tests Passed
      EREG2 = 0000

  0122 A memory location failed
      EREG2 = address of the error

MEM03  Galloping Ones/Zeros (Registers)
  0023 All Tests Passed
      EREG2 = 0000

  0123 A memory location failed
      EREG2 = address of the error

MEM04  Bits Set
  0024 No bits were set in the byte tested
      EREG2 = 0000

  0124 At least one bit was set in the byte tested
      EREG2 = number of bits set

7-60
MEM05  Checkerboard Pattern (Registers)
0025 All Tests Passed
EREG2 = 0000
0125 A memory location failed
EREG2 = address of the error

MEM06  Complementary Address
0026 All Tests Passed
EREG2 = 0000
0126 A memory location failed
EREG2 = address of the error

MEM07  Walking Ones
0027 All Tests Passed
EREG2 = 0000
0127 A memory location failed
EREG2 = address of the error

MEM08  Galloping Ones
0028 All Tests Passed
EREG2 = 0000
0128 A memory location failed
EREG2 = address of the error

MEM09  Walking Ones/Zeros
0029 All Tests Passed
EREG2 = 0000
0129 A memory location failed
EREG2 = address of the error

MEM0A  Galloping Ones/Zeros
002A All Tests Passed
EREG2 = 0000
012A A memory location failed
EREG2 = address of the error

MEM0B  Checksum
012B 16-bit Checksum is ready
EREG2 = 16-bit Checksum

MEM0C  User Pattern (Registers)
002C All Tests Passed
EREG2 = 0000
012C A memory location failed
EREG2 = address of the error

MEM0D  User Pattern
002D All Tests Passed
EREG2 = 0000
012D A memory location failed
EREG2 = address of the error
SYS01 Common Symbols
0000 No Message
    EREG2 = 0ffffh

SYS02 System Power-up
0002 All Tests Passed
    EREG2 = 0000h

0102 I/O Status Registers were unexpected
    EREG2 = IOS0 in low byte, IOS1 in high byte

0202 TIMER1 did not change over time
    EREG2 = TIMER1

0302 Zero Register was found to change
    EREG2 = Program Status Word At Failure

0402 PUSHF or POPF failed
    EREG2 = Erroneous PUSHeD or POPed value found

0502 Sticky Bit would not set
    EREG2 = 3fffh

0502 Sticky Bit would not clear
    EREG2 = 0000

0602 Carry Flag Test Failed
    EREG2 = xxxx

0702 Overflow flags would not set correctly
    EREG2 = 0002h

0702 Overflow flags would not clear correctly
    EREG2 = xxxx

0802 Interrupt Pending Register failed read/write test
    EREG2 = offending Interrupt Pending byte

SYS03 Program Counter
0003 All Tests Passed
    EREG2 = 0000

0103 Test Code Returned Early
    EREG2 = Early Time

0203 Test Code Returned Late
    EREG2 = Late Time

0303 Counter Register contained unexpected value
    EREG2 = Erroneous Counter Value
APPENDIX C
DESCRIPTION OF DIAG96.LIB BATCH FILES

The batch files that come with the library will help speed the process of either linking to the library as is, or revising library programs to suit custom purposes.

Some batch files require a parameter that provides the extensionless name of a user definable variable file to be included in the action of the batch file.

All DIAG96.LIB batch files assume that both the source and destination files reside in the same directory. Given the size of the library, and the fact that all of the files will not fit on one floppy disk, the command files will need to be edited if the user's system is not equipped with a hard disk.

INSTAL. BAT — Used to install the library on a hard disk system. To install the library, create a directory called \ DIAG96 under the main directory, insert disk 1 into drive a: and type:

    a:Instal

DST360K.BAT & DST12MEG.BAT — CAUTION: THEE BATCH FILES WILL FORMAT AND DESTROY ALL INFORMATION ON THE FLOPPIES USED. These command files were created to make the DIAG96.LIB disk set. DST360K was created for use with 360K floppy disks and requires three diskettes. DST12MEG was created for use with 1.2M disks and only needs two diskettes. The batch files will prompt you to change disks. MAKE SURE TO ENTER THE CORRECT DISK DRIVE WHEN INVOKING THESE BATCH FILES. ALSO MAKE SURE TO INCLUDE THE DRIVE ID IN THE COMMAND LINE. THESE BATCH FILES FIRST FORMAT THE DISK, AND WE ALL KNOW WHAT WHEN DOS DEFAULTS TO THE HARD DISK!!!!!!!!!! For example:

    DST12MEG a:

SCRUB.BAT — CAUTION: THIS FILE DELETES FILES USING WILDCARDS. All Diagnostic Library related files are deleted for the \ DIAG96 directory. SYS?? and MEM?? wildcards are used, so be forewarned. This batch file does not delete itself!!!! To invoke this batch file, type:

    Scrub

D96ASM.BAT — Assembles all General Diagnostic modules including the PLM compilation of D96P96.P96. To invoke the batch file, get in the \ DIAG96 directory and type:

    D96ASM

DSTASM.BAT — Assembles all Dynamic Stability Test modules. To invoke the batch file, get in \ DIAG96 directory and type:

    DSTASM

D96LP.BAT — Copies all General Diagnostic list files to a printer. Invocation must include a device where the printer resides. For example:

    D96LP lpt1

DSTLP.BAT — Copies all Dynamic Stability Test modules to a printer. Invocation must include a device where the printer resides. For example:

    DSTLP.BAT lpt1

LPONLY.BAT — Executes D96LP.BAT and DSTLP.BAT. Invocation must include a device where the printer resides. For example:

    LPONLY lpt1

7–63
D96LIB.BAT — Deletes the current DIAG96.LIB collection. Also creates a new library of the same name using the files resident in the \ DIAG96 directory bearing the General Diagnostics names. The DST96.LIB is not altered, and is included in the new DIAG96.LIB. To invoke the batch file, get in the \ DIAG96 directory and type:

D96LIB

DSTLIB.BAT — Deletes the current DST96.LIB collection. Also creates a new library of the same name using the files resident in the \ DIAG96 directory bearing the Dynamic Stability Test names. Since DST96.LIB is included in DIAG96.LIB, DIAG96.LIB is recreated by an invocation of D96LIB.BAT. To invoke this batch file, get in the \ DIAG96 directory and type:

DSTLIB

DSTRL.BAT — This batch file is of most interest to Dynamic Stability Test users. It links a specified main task to the library. This file makes assumptions about the hardware memory implementation that may not be correct. Therefore minor changes may need to be made to the DSTRL.BAT RL96 invocation statement. A file name without extension must be provided and that file must reside in the \ DIAG96 directory. The batch file assumes that the extension of the object file to be linked to the library is .OBJ. For example:

DSTRL Example_task

BLASTP.BAT — This batch file assembles the specified input file, then executes D96ASM.BAT, DSTASM.BAT, LONLY.BAT, DSTLIB.BAT, and DSTRL.BAT. Then, the listfile output of the user's assembly and the print file of the linkage are copied to the printer specified. The batch file assumes that the input file is in the \ DIAG96 directory and has a .A96 extension. For example:

BLASTP Example_lptl

BLASTN.BAT — This batch file executes all assemblies, compilations, and linkages executed in BLASTP.BAT, but no copies are sent to the printer. The batch file assumes that the input file is in the \ DIAG96 directory and has a .A96 extension. For example:

BLASTN Example_task

REGEN.BAT — Used to regenerate the library when only one module has changed. Specify the module that has changed when invoking this batch file. For example:

REGEN ALU03

MAKPLM.BAT — Used to make an impostor PLM96.LIB. The library created in not a real PLM96.LIB, and will not work with PLM programs. However, it is what is needed to use DIAG96.LIB. To invoke this batch file, get in the \ DIAG96 directory and type:

MAKPLM

MAKBH.BAT — Used to modify the library to run in an 8X9XBH. The 8X9XBH fails a flag test because of the -90 bug relating to the Z flag on add and subtract with carry is inadvertently verified by a library test. To invoke this batch file, get in the \ DIAG96 directory and type:

MAKBH

D96RL.BAT — A generalized command that links target modules to DIAG96.LIB. It is intended for used when only the General Diagnostics are being used. Provide the target object file name and the directory in which it resides. For example:

D96RL \ SOURCE \ Example_
SERIES-III MCS-96 MACRO ASSEMBLER. V1.0

SOURCE FILE: :F5:D96A96.A96
OBJECT FILE: :F5:D96A96.OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND: NOGEN DEBUG

ERR LOC OBJECT

LINE SOURCE STATEMENT
1 2 ;*******************************************************************************
3 ALL_TESTS ASM96 ;*******************************************************************************
4 #STACKSIZE(20) 0830
5 ; in order to run this module, the STACK must be ALL external, and the
6 ; data ram partitioned for memory test must not include ANY of the STACK
7 ; To call this module
8 9
10 PUSH #<RAM segment1 start address>
11 PUSH #<RAM segment1 ending address>
12 PUSH #<RAM segment2 start address>
13 PUSH #<RAM segment2 ending address>
14 PUSH #<RAM segment3 seed>
15 PUSH #<number of cycles desired for random test>
16 PUSH #<address of the last byte of rom>
17 PUSH #<an argument for mul/div tests>
18 PUSH #<a second argument for mul/div tests>
19 PUSH #<a bit pattern for memory tests>
20 CALL D96A96
21 ; Remember, this test will take a long time if large memory regions are
22 ; partitioned, or if a large number of cycles of random test numbers is
23 ; requested. For example, with 8Kbytes of Ram in each region the test
24 ; executes in 3 hours.
25 ; It is suggested that for large memory tests, that the complimentary
26 ; address test be done on the whole region at once. Then, the more
27 ; exhaustive tests done on each memory chip in the system.
28 ;*******************************************************************************
29 rseg
30 extrn sp,ereql,ereq2
31
32 cseg at 3000h
33 extrn sys01,sys02,sys03,alu01,alu02,alu03,alu04,alu05
34 extrn mem01,mem02,mem03,mem04,mem05,mem06,mem07,mem08
35 extrn mem09,mem0a,mem0b,mem0c,mem0d
36 PUBLIC D96A96
37 $eject
38 ;*******************************************************************************
39 $include (:f3:dstmac.inc) ;provides the macro BR ON Error
40 41
42 43
44 45
46 47
ERR LOC OBJECT

0000 E

0000 0000 CALL sys02
106 D96A96:
107 ;CALL the System Power Up Test
108 ;
109 110 BR_ON_ERR Error_Found
111 ;CALL the Add/Subtract test
112 ;
113 114
115 CALL alu01
116 ;
117 118 BR_ON_ERR Error_Found
119 ;CALL the MULUB test
120 ;
121 122 CALL alu02
123 ;
124 125 BR_ON_ERR Error_Found
126 ;CALL the Multiply/Divide Table
127 ;driven test
128 ;
000A E

000A 0000 CALL alu03
129 ;
130 131 BR_ON_ERR Error_Found
132 ;PUSH a random seed
133 ;
134 135 CALL alu04
136 ;PUSH the number of tests desired
137 ;;CALL the Multiply/Divide Random test
138 ;
000E E

000E 0000 CALL alu05
139 ;
140 141 BR_ON_ERR Error_Found
142 ;PUSH an argument
143 ;
144 145 PUSH 06h[sp]
146 ;PUSH another argument
147 ;;CALL the Multiply/Divide Core Test
148 ;
0012 C

0012 CB000C E

0012 0000 PUSH 0ch[sp]
149 ;
150 151 CALL alu05
152 ;PUSH a random seed
153 ;
154 155 CALL mem01
156 ;PUSH the number of tests desired
157 ;;CALL the Multiply/Divide Random test
158 ;
0016 E

0016 0000 CALL alu05
159 ;
160 161 BR_ON_ERR Error_Found
162 ;CALL a Complementary Address test
163 ;on the internal registers
164 ;
001A E

001A 0000 CALL mem02
165 ;
166 167 BR_ON_ERR Error_Found
168 ;CALL a Walking 1s/0s test on
169 ;the internal registers
170 ;
001E E

001E 0000 CALL mem03
171 ;
172 173 BR_ON_ERR Error_Found
174 ;CALL a Galloping 1s/0s test on
175 ;the internal registers
176 ;
0022 C

0022 CB000C E

0022 0000 PUSH 06h[sp]
177 ;
178 179 CALL mem04
180 ;PUSH a zero
181 ;;CALL the Bits Set Test
182 ;
0026 E

0026 0000 CALL mem05
183 ;
184 185 BR_ON_ERR Error_Found
186 ;CALL a Checkerboard Pattern test
187 ;for internal registers
188 ;
002A C

002A CB000C E

002A 0000 PUSH 14h[sp]
189 ;
190 191 CALL mem06
192 ;; and the end address of a RAM area
193 ;; and CALL a Complementary Address test
194 ;
ERR LOC OBJECT LINE SOURCE STATEMENT

008C CB0010 E 199 BR_ON_ERR Error_Found
008F CB0010 E 200 PUSH 10h[sp]
0092 EF0000 E 201 CALL mem06

202
203
204 ERR Error_Found
009C CB0014 E 208 PUSH 14h[sp]
009F CB0014 E 209 PUSH 14h[sp]
00A2 EF0000 E 210 CALL mem07

211
212
213 ERR Error_Found
00AC CB0010 E 217 PUSH 10h[sp]
00AF CB0010 E 218 PUSH 10h[sp]
00B2 EF0000 E 219 CALL mem07

220
221
222 ERR Error_Found
00BC CB0014 E 226 PUSH 14h[sp]
00BF CB0014 E 227 PUSH 14h[sp]
00C2 EF0000 E 228 CALL mem08

229
230
231 ERR Error_Found
00CC CB0010 E 236 PUSH 10h[sp]
00CF CB0010 E 237 PUSH 10h[sp]
00D2 EF0000 E 238 CALL mem08

239
240
241 ERR Error_Found
00DC CB0014 E 245 PUSH 14h[sp]
00DF CB0014 E 246 PUSH 14h[sp]
00E2 EF0000 E 247 CALL mem09

248
249
250 ERR Error_Found
00EC CB0010 E 255 PUSH 10h[sp]
00EF CB0010 E 256 PUSH 10h[sp]
00F2 EF0000 E 257 CALL mem09

258
259
260 ERR Error_Found
00FC CB0014 E 264 PUSH 14h[sp]
00FF CB0014 E 265 PUSH 14h[sp]
0102 EF0000 E 266 CALL mem0a

267
268
269 ERR Error_Found
010C CB0010 E 274 PUSH 10h[sp]
010F CB0010 E 275 PUSH 10h[sp]
0112 EF0000 E 276 CALL mem0a

277
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>011A</td>
<td>CB0002</td>
<td>278</td>
<td>BR_ON_ERR Error_Found</td>
</tr>
<tr>
<td>011D</td>
<td>EF0000</td>
<td>282</td>
<td>PUSH 02h[sp] ; PUSH a bit pattern to use and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>283</td>
<td>CALL mem0c ; CALL the Checkerboard Pattern test</td>
</tr>
<tr>
<td></td>
<td></td>
<td>284</td>
<td>; for internal registers</td>
</tr>
<tr>
<td>0125</td>
<td>CB0014</td>
<td>287</td>
<td>BR_ON_ERR Error_Found</td>
</tr>
<tr>
<td>0128</td>
<td>CB0014</td>
<td>291</td>
<td>PUSH 14h[sp] ; PUSH the start and end address</td>
</tr>
<tr>
<td>012B</td>
<td>CB0006</td>
<td>292</td>
<td>14h[sp] ; of a region of RAM, then</td>
</tr>
<tr>
<td>012E</td>
<td>EF0000</td>
<td>293</td>
<td>PUSH 06h[sp] ; PUSH a bit pattern to use, then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>294</td>
<td>CALL mem0d ; CALL the Checkerboard Pattern test</td>
</tr>
<tr>
<td></td>
<td></td>
<td>295</td>
<td>; for external memory</td>
</tr>
<tr>
<td>0136</td>
<td>CB0010</td>
<td>296</td>
<td>BR_ON_ERR Error_Found</td>
</tr>
<tr>
<td>0139</td>
<td>CB0010</td>
<td>297</td>
<td>PUSH 10h[sp] ; PUSH the start and end address</td>
</tr>
<tr>
<td>013C</td>
<td>CB0006</td>
<td>301</td>
<td>10h[sp] ; of another region of RAM, then</td>
</tr>
<tr>
<td>013F</td>
<td>EF0000</td>
<td>302</td>
<td>PUSH 06h[sp] ; PUSH a bit pattern to use, then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>303</td>
<td>CALL mem0d ; CALL the Checkerboard Pattern test</td>
</tr>
<tr>
<td></td>
<td></td>
<td>304</td>
<td>; for external memory</td>
</tr>
<tr>
<td>0147</td>
<td>CB0014</td>
<td>307</td>
<td>BR_ON_ERR Error_Found</td>
</tr>
<tr>
<td>014A</td>
<td>CB0014</td>
<td>311</td>
<td>PUSH 14h[sp] ; PUSH a starting address, and</td>
</tr>
<tr>
<td>014D</td>
<td>EF0000</td>
<td>312</td>
<td>14h[sp] ; PUSH an ending address for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>313</td>
<td>; the Program Counter Module</td>
</tr>
<tr>
<td>0155</td>
<td>CB0010</td>
<td>314</td>
<td>BR_ON_ERR Error_Found</td>
</tr>
<tr>
<td>0158</td>
<td>CB0010</td>
<td>315</td>
<td>PUSH 10h[sp] ; PUSH the start and end addresses</td>
</tr>
<tr>
<td>015B</td>
<td>EF0000</td>
<td>316</td>
<td>10h[sp] ; for a second test region for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>317</td>
<td>; the Program Counter Module</td>
</tr>
<tr>
<td>0163</td>
<td>CB0020</td>
<td>320</td>
<td>BR_ON_ERR Error_Found</td>
</tr>
<tr>
<td>0166</td>
<td>CB000A</td>
<td>321</td>
<td>PUSH #2000h ; PUSH the code starting address</td>
</tr>
<tr>
<td>0169</td>
<td>EF0000</td>
<td>322</td>
<td>PUSH 0ah[sp] ; PUSH the ending code address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>323</td>
<td>CALL mem0b ; CALL the Checksum routine</td>
</tr>
<tr>
<td>016C</td>
<td>A130000</td>
<td>330</td>
<td>LD EREG1,#0030h ; ALL DIAG96 TESTS PASSED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>331</td>
<td>; load the appropriate error code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>332</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>333</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>334</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>335</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>336</td>
<td></td>
</tr>
<tr>
<td>0170</td>
<td>CF0014</td>
<td>340</td>
<td>POP 14h[sp] ; clean off the stack</td>
</tr>
<tr>
<td>0173</td>
<td>65120000</td>
<td>343</td>
<td>ADD sp,#12h ; return to the calling program</td>
</tr>
<tr>
<td>0177</td>
<td>F0</td>
<td>344</td>
<td></td>
</tr>
<tr>
<td>0178</td>
<td></td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>0178</td>
<td>CF0014</td>
<td>346</td>
<td></td>
</tr>
<tr>
<td>017B</td>
<td>65120000</td>
<td>347</td>
<td></td>
</tr>
<tr>
<td>017F</td>
<td>F0</td>
<td>348</td>
<td></td>
</tr>
<tr>
<td>0180</td>
<td></td>
<td>349</td>
<td></td>
</tr>
<tr>
<td>NAME</td>
<td>VALUE</td>
<td>ATTRIBUTES</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------</td>
<td>--------------------------</td>
<td></td>
</tr>
<tr>
<td>ALL_TESTS_ASM96</td>
<td>------</td>
<td>MODULE STACKSIZE(20)</td>
<td></td>
</tr>
<tr>
<td>ALU01</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>ALU02</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>ALU03</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>ALU04</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>ALU05</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>BR_ON_ERR</td>
<td>------</td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>D96A96</td>
<td>0000H</td>
<td>CODE REL PUBLIC ENTRY</td>
<td></td>
</tr>
<tr>
<td>EREC1</td>
<td>------</td>
<td>REG EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>EREC2</td>
<td>------</td>
<td>REG EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>ERROR_FOUND</td>
<td>0178H</td>
<td>CODE REL ENTRY</td>
<td></td>
</tr>
<tr>
<td>MACRO_TEMP</td>
<td>0000H</td>
<td>REG REL BYTE</td>
<td></td>
</tr>
<tr>
<td>MEM01</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM02</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM03</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM04</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM05</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM06</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM07</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM08</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM09</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM0A</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM0B</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM0C</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>MEM0D</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>RESET_WATCHDOG</td>
<td>------</td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>RI</td>
<td>0006H</td>
<td>NULL ABS</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>------</td>
<td>REG EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>SP STAT</td>
<td>------</td>
<td>REG EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>SPSTATUS</td>
<td>------</td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>SPWAIT</td>
<td>------</td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>SYS01</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>SYS02</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>SYS03</td>
<td>------</td>
<td>CODE EXTERNAL</td>
<td></td>
</tr>
<tr>
<td>TI</td>
<td>0005H</td>
<td>NULL ABS</td>
<td></td>
</tr>
<tr>
<td>ZERO</td>
<td>------</td>
<td>REG EXTERNAL</td>
<td></td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETED,  NO ERROR(S) FOUND.
SERIES-III PL/M-96 V1.0 COMPILATION OF MODULE ALLDIAG96TESTS
OBJECT MODULE PLACED IN :F2:D96P96.OBJ
COMPILER INVOKED BY: PLM96.86 :F2:D96P96.P96 CODE DEBUG

1     All$Diaq96$Tests:
   DO;

2     SYS02:  PROCEDURE DWORD EXTERNAL;
3         END SYS02;

4     SYS03:  PROCEDURE (parm1,parm2) DWORD EXTERNAL;
5         DECLARE (parm1,parm2) WORD;
6         END SYS03;

7     ALU01:  PROCEDURE DWORD EXTERNAL;
8         END ALU01;

9     ALU02:  PROCEDURE DWORD EXTERNAL;
10        END ALU02;

11    ALU03:  PROCEDURE DWORD EXTERNAL;
12        END ALU03;

13    ALU04:  PROCEDURE (parm1,parm2) DWORD EXTERNAL;
14        DECLARE (parm1,parm2) WORD;
15        END ALU04;

16    ALU05:  PROCEDURE (parm1,parm2) DWORD EXTERNAL;
17        DECLARE (parm1,parm2) WORD;
18        END ALU05;

19    MEM01:  PROCEDURE DWORD EXTERNAL;
20        END MEM01;

21    MEM02:  PROCEDURE DWORD EXTERNAL;
22        END MEM02;

23    MEM03:  PROCEDURE DWORD EXTERNAL;
24        END MEM03;

25    MEM04:  PROCEDURE (parm1) DWORD EXTERNAL;
26        DECLARE (parm1) WORD;
27        END MEM04;

28    MEM05:  PROCEDURE DWORD EXTERNAL;
29        END MEM05;

30    MEM06:  PROCEDURE (parm1,parm2) DWORD EXTERNAL;
31        DECLARE (parm1,parm2) WORD;
32        END MEM06;

33    MEM07:  PROCEDURE (parm1,parm2) DWORD EXTERNAL;
34        DECLARE (parm1,parm2) WORD;
35        END MEM07;
MEM08: PROCEDURE (parm1, parm2) DWord EXTERNAL;
DECLARE (parm1, parm2) WORD;
END MEM08;

MEM09: PROCEDURE (parm1, parm2) DWord EXTERNAL;
DECLARE (parm1, parm2) WORD;
END MEM09;

MEM0A: PROCEDURE (parm1, parm2) DWord EXTERNAL;
DECLARE (parm1, parm2) WORD;
END MEM0A;

MEM0B: PROCEDURE (parm1, parm2) DWord EXTERNAL;
DECLARE (parm1, parm2) WORD;
END MEM0B;

MEM0C: PROCEDURE (parm1) DWord EXTERNAL;
DECLARE (parm1) WORD;
END MEM0C;

MEM0D: PROCEDURE (parm1, parm2, parm3) DWord EXTERNAL;
DECLARE (parm1, parm2, parm3) WORD;
END MEM0D;

DECLARE result DWord;
DECLARE error$codes STRUCTURE (number WORD, detail WORD) AT (.result);

D96P96: PROCEDURE (ram1$start, ram1$stop,
ram2$start, ram2$stop,
random$seed, random$length,
top$of$code, argument1, argument2,
bit$pattern) DWord PUBLIC;

DECLARE (ram1$start, ram1$stop,
ram2$start, ram2$stop,
random$seed, random$length,
top$of$code, argument1, argument2,
bit$pattern) WORD;

result=SYS02;
IF error$codes.number > 255 THEN GOTO end$tests;

result=ALU01;
IF error$codes.number > 255 THEN GOTO end$tests;

result=ALU02;
IF error$codes.number > 255 THEN GOTO end$tests;

result=ALU03;
IF error$codes.number > 255 THEN GOTO end$tests;

result=ALU04(47efH,1000H);
71 2 IF error$codes.number > 255 THEN GOTO end$tests;
73 2 result=ALU05(argument1,argument2);
74 2 IF error$codes.number > 255 THEN GOTO end$tests;
76 2 result=MEM01;
77 2 IF error$codes.number > 255 THEN GOTO end$tests;
79 2 result=MEM02;
80 2 IF error$codes.number > 255 THEN GOTO end$tests;
82 2 result=MEM03;
83 2 IF error$codes.number > 255 THEN GOTO end$tests;
85 2 result=MEM04(0);
86 2 IF error$codes.number > 255 THEN GOTO end$tests;
88 2 result=MEM05;
89 2 IF error$codes.number > 255 THEN GOTO end$tests;
91 2 result=MEM06(ram1$start,ram1$stop);
92 2 IF error$codes.number > 255 THEN GOTO end$tests;
94 2 result=MEM06(ram2$start,ram2$stop);
95 2 IF error$codes.number > 255 THEN GOTO end$tests;
97 2 result=MEM07(ram1$start,ram1$stop);
98 2 IF error$codes.number > 255 THEN GOTO end$tests;
100 2 result=MEM07(ram2$start,ram2$stop);
101 2 IF error$codes.number > 255 THEN GOTO end$tests;
103 2 result=MEM08(ram1$start,ram1$stop);
104 2 IF error$codes.number > 255 THEN GOTO end$tests;
106 2 result=MEM08(ram2$start,ram2$stop);
107 2 IF error$codes.number > 255 THEN GOTO end$tests;
109 2 result=MEM09(ram1$start,ram1$stop);
110 2 IF error$codes.number > 255 THEN GOTO end$tests;
112 2 result=MEM09(ram2$start,ram2$stop);
113 2 IF error$codes.number > 255 THEN GOTO end$tests;
115 2 result=MEM0A(ram1$start,ram1$stop);
116 2 IF error$codes.number > 255 THEN GOTO end$tests;
118 2 result=MEM0A(ram2$start,ram2$stop);
119 2 IF error$codes.number > 255 THEN GOTO end$tests;
121 2 result=MEM0C(bit$pattern);
122 2 IF error$codes.number > 255 THEN GOTO end$tests;
124 2 result=MEM0D(ram1$start,ram1$stop.bit$pattern);
125 2 IF error$codes.number > 255 THEN GOTO end$tests;
127 2 result=MEM0D(ram2$start,ram2$stop.bit$pattern);
128 2 IF error$codes.number > 255 THEN GOTO end$tests;
130 2 result=SYS03(ram1$start,ram1$stop);
131 2 IF error$codes.number > 255 THEN GOTO end$tests;
133 2 result=SYS03(ram2$start,ram2$stop);
134 2 IF error$codes.number > 255 THEN GOTO end$tests;
136 2 result=MEM0B(2080h,top$of$code);
137 2 error$codes.number=00f0h;
138 2 end$tests: RETURN result;
139 2 END D96P96;
140 1 END All$Diag96$Tests;
D96P96:

; STATEMENT 56
0000 C800 E PUSH $FRAME01
0002 A01B00 E LD $FRAME01, SP
; STATEMENT 58
0005 EF0000 E CALL SYS02
0008 A01E02 R LD RESULT+2H, TMP2
000B A01C00 R LD RESULT, TMP0
; STATEMENT 59
000E 89F0000 R CMP ERRORCODES, #OFFH
0012 D102 R BH @0001
; STATEMENT 60
0014 2226 R BR ENDTST
; STATEMENT 61
0016 @0001:
0016 EF0000 E CALL ALU01
0019 A01E02 R LD RESULT+2H, TMP2
001C A01C00 R LD RESULT, TMP0
; STATEMENT 62
001F 89F0000 R CMP ERRORCODES, #OFFH
0023 D102 R BH @0002
; STATEMENT 63
0025 2215 R BR ENDTST
; STATEMENT 64
0027 @0002:
0027 EF0000 E CALL ALU02
002A A01E02 R LD RESULT+2H, TMP2
002D A01C00 R LD RESULT, TMP0
; STATEMENT 65
0030 89F0000 R CMP ERRORCODES, #OFFH
0034 D102 R BH @0003
; STATEMENT 66
0036 2204 R BR ENDTST
; STATEMENT 67
0038 @0003:
0038 EF0000 E CALL ALU03
003B A01E02 R LD RESULT+2H, TMP2
003E A01C00 R LD RESULT, TMP0
; STATEMENT 68
0041 89F0000 R CMP ERRORCODES, #OFFH
0045 D102 R BH @0004
; STATEMENT 69
0047 21F3 R BR ENDTST
; STATEMENT 70
0049 @0004:
0049 C9EF47 R PUSH #47EFH
004C C90010 R PUSH #1000H
004F EF0000 E CALL ALU04
0052 A01E02 R LD RESULT+2H, TMP2
0055 A01C00 R LD RESULT, TMP0
; STATEMENT 71
0058 89F0000 R CMP ERRORCODES, #OFFH
005C D102 R BH @0005
; STATEMENT 72
005E 21DC R BR ENDTST
; STATEMENT 73
E0005:  E PUSH ARGUMENT1[FRAME01]
E0006:  E PUSH ARGUMENT2[FRAME01]
E0008:  E CALL ALU05
R  A01C00  R LD RESULT+2H_TMP2
R  A01C00  R LD RESULT_TMP0
; STATEMENT 74
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00006
; STATEMENT 75
R  21C5  BR ENDTESTS
; STATEMENT 76
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00007
; STATEMENT 78
R  2184  BR ENDTESTS
; STATEMENT 79
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00008
; STATEMENT 80
R  21A3  BR ENDTESTS
; STATEMENT 81
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00009
; STATEMENT 82
R  2192  BR ENDTESTS
; STATEMENT 83
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00010
; STATEMENT 84
R  217F  BR ENDTESTS
; STATEMENT 85
R  C000  PUSH R0
E  EF0000  E CALL MEM01
R  A01E02  R LD RESULT+2H_TMP2
R  A01C00  R LD RESULT_TMP0
; STATEMENT 86
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00011
; STATEMENT 87
R  217F  BR ENDTESTS
; STATEMENT 88
R  EF0000  E CALL MEM02
R  A01E02  R LD RESULT+2H_TMP2
R  A01C00  R LD RESULT_TMP0
; STATEMENT 90
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00012
; STATEMENT 91
R  21A3  BR ENDTESTS
; STATEMENT 92
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00013
; STATEMENT 93
R  2184  BR ENDTESTS
; STATEMENT 94
R  C000  PUSH R0
E  EF0000  E CALL MEM03
R  A01E02  R LD RESULT+2H_TMP2
R  A01C00  R LD RESULT_TMP0
; STATEMENT 96
R  89FF0000  R CMP ERRORCODES.0FFH
BHN 00014
; STATEMENT 97
R  2192  BR ENDTESTS
; STATEMENT 98
STATEMENT 117
019B 209F
BR ENTESTS
; STATEMENT 118
019D 00014:

019D CB0012 E PUSH RAM2START[?FRAME01]
01A0 CB0010 E PUSH RAM2STOP[?FRAME01]
01A3 EF0000 E CALL MEM00
01A6 A01E02 R LD RESULT+2H,TMP2
01A9 A01C00 R LD RESULT,TMP0
; STATEMENT 119
01AC 89FF0000 R CMP ERRORCODES,#0FFH
01B0 D102 BNH @0015
; STATEMENT 120
01B2 2088 BR ENTESTS
; STATEMENT 121

01B4 00015:

01B4 CB0004 E PUSH BITPATTERN[?FRAME01]
01B7 EF0000 E CALL MEM00
01BA A01E02 R LD RESULT+2H,TMP2
01BD A01C00 R LD RESULT,TMP0
; STATEMENT 122
01C0 89FF0000 R CMP ERRORCODES,#0FFH
01C4 D102 BNH @0016
; STATEMENT 123
01C6 2074 BR ENTESTS
; STATEMENT 124

01C8 00016:

01C8 CB0016 E PUSH RAM1START[?FRAME01]
01CB CB0014 E PUSH RAM1STOP[?FRAME01]
01CE CB0004 E PUSH BITPATTERN[?FRAME01]
01D1 EF0000 E CALL MEM00
01D4 A01E02 R LD RESULT+2H,TMP2
01D7 A01C00 R LD RESULT,TMP0
; STATEMENT 125
01DA 89FF0000 R CMP ERRORCODES,#0FFH
01DE D102 BNH @0017
; STATEMENT 126
01E0 205A BR ENTESTS
; STATEMENT 127

01E2 00017:

01E2 CB0012 E PUSH RAM2START[?FRAME01]
01E5 CB0010 E PUSH RAM2STOP[?FRAME01]
01E8 CB0004 E PUSH BITPATTERN[?FRAME01]
01EB EF0000 E CALL MEM00
01EE A01E02 R LD RESULT+2H,TMP2
01F1 A01C00 R LD RESULT,TMP0
; STATEMENT 128
01F4 89FF0000 R CMP ERRORCODES,#0FFH
01FB D102 BNH @0018
; STATEMENT 129
01FA 2040 BR ENTESTS
; STATEMENT 130

01FC 00018:

01FC CB0016 E PUSH RAM1START[?FRAME01]
01FF CB0014 E PUSH RAM1STOP[?FRAME01]
0202 EF0000 E CALL SYS03
0205 A01E02 R LD RESULT+2H,TMP2
0208 A01C00 R LD RESULT,TMP0
; STATEMENT 131
020B 09FF0000 R CMP ERRORCODES,#0FFH
020F D182 BNH @0019
; STATEMENT 132
0211 2029 BR ENDTESTS
; STATEMENT 133
0213 0019:
0213 CB0012 E PUSH RAM2START[?FRAME01]
0216 CB0010 E PUSH RAM2STOP[?FRAME01]
0219 EF0000 E CALL SYS03
021C A01E02 R LD RESULT+2H,TMP2
021F A01C00 R LD RESULT,TMP0
; STATEMENT 134
0222 09FF0000 R CMP ERRORCODES,#0FFH
0226 D182 BNH @001A
; STATEMENT 135
0228 2012 BR ENDTESTS
; STATEMENT 136
022A 001A:
022A C90020 E PUSH #2000H
022D CB000A E PUSH TOPOFCODE[?FRAME01]
0230 EF0000 E CALL MEM06
0233 A01E02 R LD RESULT+2H,TMP2
0236 A01C00 R LD RESULT,TMP0
; STATEMENT 137
0239 AD0000 R LDBZE ERRORCODES,#0F0H
; STATEMENT 138
023C ENDTESTS:
023C A0021E R LD TMP2,RESULT+2H
023F A0001C R LD TMP0,RESULT
0242 CC00 E POP ?FRAME01
0244 A21822 LD TMP6,[SP]
0247 6516001B ADD SP,#16H
024B E222 BR [TMP6]
; STATEMENT 139
; STATEMENT 140
024D END

MODULE INFORMATION:

| CODE AREA SIZE | = 024DH 589D |
| CONSTANT AREA SIZE | = 0000H 6D |
| DATA AREA SIZE | = 0000H 6D |
| STATIC REGS AREA SIZE | = 0004H 4D |
| OVERLAYABLE REGS AREA SIZE | = 0000H 6D |
| MAXIMUM STACK SIZE | = 000AH 10D |

PL/M-96 COMPIILATION COMPLETE. 0 WARNINGS, 0 ERRORS
**MCS-96 MACRO ASSEMBLER  SELECTED_TESTS_ASM96**

**SERIES-III MCS-96 MACRO ASSEMBLER, V1.0**

**SOURCE FILE:** :F5:D96FST.A96  
**OBJECT FILE:** :F5:D96FST.OBJ  
**CONTROLS SPECIFIED IN INVOCATION COMMAND:** NOGEN DEBUG

<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>rseg</td>
<td>27</td>
<td>extrn sp,ereg1,ereg2</td>
</tr>
<tr>
<td>0000</td>
<td>cseg</td>
<td>31</td>
<td>extrn sys01.sys02.sys03.sys04.sys05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>extrn alu01.alu02.alu03.alu04.alu05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>extrn mem01.mem02.mem03.mem04.mem05.mem06.mem07.mem08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>extrn mem09.mem10.mem11.mem12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PUBLIC D96FST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$eject</td>
</tr>
</tbody>
</table>

In order to run this module, the STACK must be ALL external, and the data ram partitioned for memory test must not include ANY of the STACK. To call this module:

```asm
PUSH #<RAM segment1 start address>
PUSH #<RAM segment1 ending address>
PUSH #<RAM segment2 start address>
PUSH #<RAM segment2 ending address>
PUSH #<random seed>
PUSH #<number of cycles desired for random test>
PUSH #<address of the last byte of rom>
PUSH #<an argument for mul/div tests>
PUSH #<a second argument for mul/div tests>
PUSH #<a bit pattern for memory tests>
CALL D96FST
```

**$eject**
ERR LOC  OBJECT

LINE                      SOURCE STATEMENT
38      39 $include (:f3:dstmac.inc)
48      41 ;**************************************************
41      42 ;**************************************************
42      43 rseq
46      44 extern zero,sp_stat
47      45 macro_temp: DSR 1
48      46 extrn zero,sp_stat
49      47 cseg
50      48 ti equ 5
51      49 ri equ 6
52      50 ti equ 5
51      51 ri equ 6
52      52 ;**************************************************
53      53 ;**************************************************
54      54 ;**************************************************
55      55 RESET_WATCHDOG MACRO ;macro to reset the watchdog
56      56 LDB 0ah, #leh
57      57 LDB 0ah, #0eh
58      58 ENDM
59      59 ;**************************************************
60      60 ;**************************************************
61      61 ;**************************************************
62      62 ;**************************************************
63      63 SPSTATUS MACRO vl ; macro to read sp_stat to
64      64 LOCAL Sp_read ; work around the ri/ti bug
65      65 Sp_read:
66      66 LDB macro_temp,sp_stat
67      67 ORB vl,macro_temp
68      68 ANDB macro_temp,$0110000B
69      69 JNE Sp_read
70      70 ENDM
71      71 ;**************************************************
72      72 ;**************************************************
73      73 ;**************************************************
74      74 ;**************************************************
75      76 SPWAIT MACRO v2 ; macro to wait for ri/ti set
76      77 ; and avoid 8x9x-90 bug.
77      78 JBC sp_stat,v2.$ ; with a full duplex line.
78      79 LDB zero,sp_stat
79      80 ;**************************************************
80      81 ;**************************************************
81      82 ENDM
82      83 ;**************************************************
83      84 ;**************************************************
84      85 ;**************************************************
85      86 ;**************************************************
86      87 BR_ON_ERR MACRO Label ; macro to test high byte of
87      88 CMPB eregi+1h,zero ; EREGI and branch away if
89      89 ; the byte is non-zero (which
90      90 ; means there was an error).
90      91 BNE Label
91      92 ENDM
92      93 ;**************************************************
93      94 ;**************************************************
94
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>96</td>
<td>D96FST:</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>97</td>
<td>CALL sys02.A96</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>98</td>
<td>;CALL the Power Up Test</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>99</td>
<td>100</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>100</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>101</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>102</td>
<td>;CALL the Add/Subtract test</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>103</td>
<td>CALL alu01</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>104</td>
<td>105</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>106</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>107</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>108</td>
<td>;CALL the MULUB test</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>109</td>
<td>CALL alu02</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>112</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>113</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>114</td>
<td>;CALL the Multiply/Divide Table driven test</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>115</td>
<td>CALL alu03</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>118</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>119</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>120</td>
<td>;CALL the Multiply/Divide Random test</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>121</td>
<td>CALL alu05</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>122</td>
<td>123</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>124</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>125</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>EF0000</td>
<td>126</td>
<td>;PUSH a seed and test length</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>127</td>
<td>PUSH 0ch[sp]</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>128</td>
<td>;PUSH an argument</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>129</td>
<td>PUSH 06h[sp]</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>130</td>
<td>;PUSH another argument</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>131</td>
<td>CALL alu05</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>132</td>
<td>;CALL the Multiply/Divide Core test</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>133</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>134</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>135</td>
<td>;CALL the Complementary Address test for internal registers</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>136</td>
<td>CALL mem01</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>137</td>
<td>138</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>139</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>140</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>141</td>
<td>;CALL the Galloping Is/0s test for internal registers</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>142</td>
<td>CALL mem03</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>143</td>
<td>144</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>145</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>146</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>147</td>
<td>;CALL the Chekerboard Pattern test for internal registers</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>148</td>
<td>CALL mem05</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>149</td>
<td>150</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>151</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>152</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>153</td>
<td>;CALL the Chekerboard Pattern test for external RAM</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>154</td>
<td>CALL mem05</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>155</td>
<td>156</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>157</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>158</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>159</td>
<td>;CALL the Complementary Address test for external RAM</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>160</td>
<td>CALL mem06</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>161</td>
<td>162</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>163</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>164</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>165</td>
<td>;PUSH a start and end address</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>166</td>
<td>PUSH 14h[sp]</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>167</td>
<td>;for a region of RAM to conduct</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>168</td>
<td>CALL mem06</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>169</td>
<td>;the Complementary Address test for</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>170</td>
<td>171</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>172</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>173</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>174</td>
<td>;for another region of RAM to conduct</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>175</td>
<td>CALL mem06</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>176</td>
<td>;the Complementary Address test for</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>177</td>
<td>178</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>179</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>180</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>181</td>
<td>;for external RAM</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>182</td>
<td>CALL mem06</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>183</td>
<td>184</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>185</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>186</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>187</td>
<td>;for another region of RAM to conduct</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>188</td>
<td>CALL mem06</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>189</td>
<td>;the Complementary Address test for</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>190</td>
<td>191</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>192</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>193</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>194</td>
<td>;for external RAM</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>195</td>
<td>CALL mem06</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>196</td>
<td>197</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>198</td>
<td>BR_ON_ERR</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>199</td>
<td>Error_Found</td>
</tr>
<tr>
<td>0000</td>
<td>CB0010</td>
<td>200</td>
<td>;for a region of RAM, and PUSH</td>
</tr>
</tbody>
</table>
ERR LOC OBJECT LINE SOURCE STATEMENT
0088 CB0006 E 186 PUSH 06h[sp] ; a bit pattern to use in the
0089 EB0000 E 187 CALL mem0d ; Checkerboard Pattern test for
188 external RAM
189 BR_ON_ERR Error_Found
0091 CB0010 E 193 PUSH 10h[sp] ; PUSH a start and end address for
0095 CB0010 E 194 PUSH 10h[sp] ; another region of RAM, and PUSH
0099 CB0006 E 195 PUSH 06h[sp] ; a bit pattern to use in the
009C EF0000 E 196 CALL mem0d ; Checkerboard Pattern test for
197 external RAM
198 BR_ON_ERR Error_Found
00A4 CB0014 E 203 PUSH 14h[sp] ; PUSH a start and end address
00A7 CB0014 E 204 PUSH 14h[sp] ; for a region of RAM to conduct
00AA EF0000 E 205 CALL sys03 ; the Program Counter test
206 BR_ON_ERR Error_Found
00B2 CB0010 E 212 PUSH 10h[sp] ; PUSH a start and end address
00B5 CB0010 E 213 PUSH 10h[sp] ; for another region of RAM to conduct
00B8 EF0000 E 214 CALL sys03 ; the Program Counter test
215 BR_ON_ERR Error_Found
00C0 C90200 E 221 PUSH #2008h ; PUSH the code starting address
00C3 CB000A E 222 PUSH 0Ah[sp] ; PUSH the code ending address
00C6 EF0000 E 223 CALL mem0b ; CALL the checksum routine
00C9 A310000 E 224 LD eregl,#0031h ; ALL DIAG96 TESTS PASSED
225 ; load appropriate error code
00CD CF0014 E 226 POP 14h[sp] ; clean off the stack
00D0 65120000 E 227 ADD sp,#12h ; return to the calling program
00D4 F0 228 RET
00D5 231 RET
00D5 CF0014 E 232 Error_Found:
00DD 233 Error_Found:
00DD CF0014 E 234 POP 14h[sp] ; clean off the stack
00DD 65120000 E 235 ADD sp,#12h ; return to the calling program
00DD F0 236 RET
00DD 237 RET
00DD 238 RET
00DD 239 ;*********************************************************************
00DD 240 end
<table>
<thead>
<tr>
<th>N A M E</th>
<th>VALUE</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU01</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>ALU02</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>ALU03</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>ALU04</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>ALU05</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>BR_ON_ERR</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>D96FST</td>
<td>0000H</td>
<td>CODE REL PUBLIC ENTRY</td>
</tr>
<tr>
<td>EREG1</td>
<td>------</td>
<td>REG EXTERNAL</td>
</tr>
<tr>
<td>EREG2</td>
<td>------</td>
<td>REG EXTERNAL</td>
</tr>
<tr>
<td>ERROR_FOUND</td>
<td>0055H</td>
<td>CODE REL ENTRY</td>
</tr>
<tr>
<td>MACRO_TEMP</td>
<td>0000H</td>
<td>REG REL BYTE</td>
</tr>
<tr>
<td>MEM01</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM02</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM03</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM04</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM05</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM06</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM07</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM08</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM09</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM10</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM11</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM12</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM13</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM14</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM15</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM16</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM17</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM18</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM19</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>MEM20</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>RI</td>
<td>0006H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>SELECTED_TESTS_ASM96</td>
<td>------</td>
<td>MODULE STACKSIZE(20)</td>
</tr>
<tr>
<td>SP</td>
<td>------</td>
<td>REG EXTERNAL</td>
</tr>
<tr>
<td>SP_STAT</td>
<td>------</td>
<td>REG EXTERNAL</td>
</tr>
<tr>
<td>SP_STATUS</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>SP_WAKE</td>
<td>------</td>
<td>MACRO</td>
</tr>
<tr>
<td>SYS01</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>SYS02</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>SYS03</td>
<td>------</td>
<td>CODE EXTERNAL</td>
</tr>
<tr>
<td>TI</td>
<td>0005H</td>
<td>NULL ABS</td>
</tr>
<tr>
<td>ZERO</td>
<td>------</td>
<td>REG EXTERNAL</td>
</tr>
</tbody>
</table>

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.
MCS-96 MACRO ASSEMBLER DSTUSR

SOURCE FILE: :F2:DSTUSR.A96
OBJECT FILE: :F2:DSTUSR.OBJ
CONTROLS SPECIFIED IN INVOCATION COMMAND: GEN DEBUG

ERR LOC OBJECT LINE SOURCE STATEMENT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0040</td>
<td>0040</td>
<td>0040</td>
<td>0000</td>
</tr>
<tr>
<td>0040</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>4200</td>
<td>4200</td>
</tr>
<tr>
<td>2080</td>
<td>2080</td>
<td>2080</td>
<td>2080</td>
</tr>
<tr>
<td>2080</td>
<td>2084</td>
<td>2084</td>
<td>2087</td>
</tr>
<tr>
<td>2088</td>
<td>2088</td>
<td>2088</td>
<td>2088</td>
</tr>
<tr>
<td>20AC</td>
<td>20AC</td>
<td>20AC</td>
<td>20AC</td>
</tr>
</tbody>
</table>

;***********************************************************************
DSTUSR MODULE main.stacksize(2)
;**********************************************************************
oseg at 40h
User Registers: DSB 1ch
temp set User Registers:WORD
rseq
EXTRN sp.zero.timer1.eregl

dseg at 100h
DSEG1: DSB 700h
;to ensure that the STACK does not get
;located in an area of RAM that will be
;memory tested, reserve those regions
;as data segments.

DSEG2: DSB 1e00h

cseg, at 2080h
extrn alu04,alu01,alu02,mem06,mem0a,error_proc.alu05
EXTRN DSTISR

;wait for sbe96 NMIs to stop
LD temp,#0ffh
DJNZ temp.S

;RAM segment1 start address
;RAM segment1 end address
;RAM segment2 start address
;RAM segment2 end address
;random seed
;random test length
;top of code address
;an argument for mul/div test
;another argument for mul/div test
;bit pattern for memory test
;CALL the Dynamic Stability Test
;initialization routine

;use the multiply/divide core
test on the arguments
<table>
<thead>
<tr>
<th>ERR LOC</th>
<th>OBJECT</th>
<th>LINE</th>
<th>SOURCE</th>
<th>STATEMENT</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>20B5</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1,zero ; #8000h and #8000h in all combinations</td>
<td></td>
</tr>
<tr>
<td>20B8</td>
<td>D022074</td>
<td></td>
<td>bne</td>
<td>error_found ; combinations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IJE</td>
<td>$+4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ISJMP</td>
<td>error_found ; combinations</td>
<td></td>
</tr>
<tr>
<td>20BC</td>
<td>C90000</td>
<td>E</td>
<td>push</td>
<td>#8000h</td>
<td></td>
</tr>
<tr>
<td>20BF</td>
<td>C90000</td>
<td></td>
<td>push</td>
<td>#8000h</td>
<td></td>
</tr>
<tr>
<td>20C2</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>alu05</td>
<td></td>
</tr>
<tr>
<td>20C5</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1,zero</td>
<td></td>
</tr>
<tr>
<td>20C8</td>
<td>D022064</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IJE</td>
<td>$+4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ISJMP</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td>20CC</td>
<td>C90000</td>
<td></td>
<td>push</td>
<td>#8000h</td>
<td></td>
</tr>
<tr>
<td>20CF</td>
<td>C90000</td>
<td></td>
<td>push</td>
<td>#8000h</td>
<td></td>
</tr>
<tr>
<td>20D2</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>alu05</td>
<td></td>
</tr>
<tr>
<td>20D5</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1,zero</td>
<td></td>
</tr>
<tr>
<td>20D8</td>
<td>D756</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td>20DA</td>
<td>C90000</td>
<td></td>
<td>push</td>
<td>#8000h</td>
<td></td>
</tr>
<tr>
<td>20DD</td>
<td>C90000</td>
<td></td>
<td>push</td>
<td>#8000h</td>
<td></td>
</tr>
<tr>
<td>20E0</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>alu05</td>
<td></td>
</tr>
<tr>
<td>20E3</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1,zero</td>
<td></td>
</tr>
<tr>
<td>20E6</td>
<td>D748</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td>20EB</td>
<td>C90001</td>
<td></td>
<td>push</td>
<td>#100h</td>
<td></td>
</tr>
<tr>
<td>20EB</td>
<td>C9FF07</td>
<td></td>
<td>push</td>
<td>#7ffh</td>
<td></td>
</tr>
<tr>
<td>20EE</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>mem0a</td>
<td></td>
</tr>
<tr>
<td>20F1</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1,zero</td>
<td></td>
</tr>
<tr>
<td>20F4</td>
<td>D73A</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td>20F6</td>
<td>C800</td>
<td>E</td>
<td>push</td>
<td>timer1</td>
<td></td>
</tr>
<tr>
<td>20F8</td>
<td>C90020</td>
<td></td>
<td>push</td>
<td>#2000h</td>
<td></td>
</tr>
<tr>
<td>20FB</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>alu04</td>
<td></td>
</tr>
<tr>
<td>20FE</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1h,zero</td>
<td></td>
</tr>
<tr>
<td>2101</td>
<td>D72D</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td>2103</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>alu01</td>
<td></td>
</tr>
<tr>
<td>2106</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1h,zero</td>
<td></td>
</tr>
<tr>
<td>2109</td>
<td>D725</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td>210B</td>
<td>C90042</td>
<td></td>
<td>push</td>
<td>#4200h</td>
<td></td>
</tr>
<tr>
<td>210E</td>
<td>C9FF5F</td>
<td></td>
<td>push</td>
<td>#5ffh</td>
<td></td>
</tr>
<tr>
<td>2111</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>mem06</td>
<td></td>
</tr>
<tr>
<td>2114</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1,zero</td>
<td></td>
</tr>
<tr>
<td>2117</td>
<td>D717</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td>2119</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>alu02</td>
<td></td>
</tr>
<tr>
<td>211C</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1h,zero</td>
<td></td>
</tr>
<tr>
<td>211F</td>
<td>D70F</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2121</td>
<td>C800</td>
<td>E</td>
<td>push</td>
<td>timer1</td>
<td></td>
</tr>
<tr>
<td>2123</td>
<td>C90020</td>
<td></td>
<td>push</td>
<td>#2000h</td>
<td></td>
</tr>
<tr>
<td>2126</td>
<td>EF0000</td>
<td>E</td>
<td>call</td>
<td>alu04</td>
<td></td>
</tr>
<tr>
<td>2129</td>
<td>980001</td>
<td>E</td>
<td>cmpb</td>
<td>eregl+1h,zero</td>
<td></td>
</tr>
<tr>
<td>212C</td>
<td>D702</td>
<td></td>
<td>bne</td>
<td>error_found</td>
<td></td>
</tr>
</tbody>
</table>

- Perform a galloping 1s/0s test on a small section of RAM.
- Send a timer1 based seed to the random number based multiply/divide test and let it run for a string of 2000h argument pairs.
- Perform the add/subtract test.
- Perform a Complementary address test on a large section of RAM.
- Perform the MULJUB test.
- Send another timer1 based seed to the random number based multiply/divide test.
Error found:

- if an error is found, disable
  interrupts and call the error
  procedure in the DST96.LIB.

- the test that found an error will
  have placed the appropriate
  error codes in locations EREG1 and
  EREG2 for output through Error_Proc

---

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.
80960kx Self-Test
INTRODUCTION

How do you know that your product still works? Are there diagnostics in the machine? If so how do you know the embedded controller itself works properly? These questions point up a very important area in embedded control—diagnostics—the task that can help avoid damaging situations or expensive down time. Often the embedded control designer includes diagnostics in the boot-up routines. The diagnostic test is written to verify system functionality but cannot test much, if any of the embedded processor. This leaves a gap in the diagnostic effort, and as processors become more complex the gap becomes more critical. The 80960Kx and 80960MC embedded processors attempt to fill this gap by including a self-test that executes upon power-up and reset. This self-test checks out basic processor functionality and bus activity. After the self-test has completed successfully system diagnostics may be done with the knowledge that the processor is working properly.

GOALS OF SELF-TEST

The self-test was designed to guarantee that each "failure check" is within known boundaries. The self-test is executed in two phases: the first phase tristates all local-bus pins denying any bus accesses. Self-test can then proceed to test the internals of the part. The last phase of the self-test reads the first eight words in memory and performs a checksum. Failure of the test is indicated on the FAILURE pin.

Self-test also had to be very efficient in order to fit in microcode. 86 microinstructions were allocated for self-test. Carefully chosen values are written to and read from internal functional units in software loops allowing complete arrays to be tested. These 86 microinstructions execute test loops that run in 46,500 cycles.

The coverage of the self-test is about 50%. The first goal of not allowing bus accesses during self-test means the bus interface and certain internal paths cannot be tested. It also means that no indication of the test result is presented to the outside world until the first phase of the self-test is completed 46,500 cycles later. What this means is that fault grading of the self-test requires massive compute and memory resources and is simply not yet practical. Test coverage was approximated by taking the percentage of the tested die size area. Although this is a crude measurement the regular arrays are easily measured for size and tested, and since their total area is a significant portion of the die area it is felt the 50% value is a reasonable approximation at this time.

MAJOR BLOCKS OF THE 80960

The 80960KB, 80960KA, and 80960MC all share a core of internal functional blocks. The core functional blocks are: Bus Control Logic (BCL), Instruction Fetch Unit (IFU), Instruction Decoder (ID), MicroInstruction Sequencer (MIS), and Integer Execution Unit (IEU). The 80960KB adds a Floating Point Unit and the 80960MC adds both a Floating Point Unit and a Memory Management Unit (MMU).

Each of the functional blocks within the processors are listed below with transistor count percentages.

<table>
<thead>
<tr>
<th>80960KA</th>
<th>80960KB</th>
<th>80960MC</th>
<th>Function Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>7%</td>
<td>6%</td>
<td>5%</td>
<td>Bus Control Logic</td>
</tr>
<tr>
<td>24%</td>
<td>21%</td>
<td>18%</td>
<td>Instruction Fetch Unit</td>
</tr>
<tr>
<td>6%</td>
<td>5%</td>
<td>5%</td>
<td>Instruction Decoder</td>
</tr>
<tr>
<td>41%</td>
<td>35%</td>
<td>30%</td>
<td>MicroInstruction Sequencer</td>
</tr>
<tr>
<td>22%</td>
<td>19%</td>
<td>16%</td>
<td>Integer Execution Unit</td>
</tr>
<tr>
<td></td>
<td>14%</td>
<td>12%</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td></td>
<td>14%</td>
<td></td>
<td>Memory Management Unit</td>
</tr>
</tbody>
</table>

The self-test tests all of the major regular structures in the processor. These include the MIS ROM, IFU Cache (instruction cache), MMU Cache (80960MC only), and the IEU RAM array which includes the local and global registers and register cache. In addition the MIS control, logic and the IEU literals are directly tested.

There are several functions and paths in the processors that are not directly tested but are tested by indirection. The microinstruction bus; data bus; and the IEU, TLB, and IFU controls and internal buses are tested indirectly when the major blocks are tested.

Several functional blocks are simply not tested by the self-test due to the no outside bus activity restriction. These are the BCL, IFU instruction pointer and fetch control, ID, and the FPU. The BCL, IFU, and ID are not tested because it would require instructions fetched from external memory to be run through the fetch and decode stages of the pipeline violating the first goal of the self-test. The FPU is not tested because a comprehensive test would simply be too large for the size limit of self-test.

Generally, any functions due to the fetch and decode stages of the pipeline can be tested in a diagnostic test after self-test by simply executing instructions from each of the five instruction formats using various operands and types. It should be noted that each of the parts are comprehensively "fault graded" to a 97% level before they are shipped to customers. This "fault grading" has a 99.9% level of confidence.
SELF-TEST ALGORITHM

The self-test algorithm consists of writing then reading values from the arrayed structures in the 80960Kx. The values chosen test the logic for “stuck-ats”. These faults manifest themselves by being fixed at a certain value, for example a node that “opens” by electromigration and is subsequently pulled low by capacitive coupling to a nearby VSS node is “stuck at” VSS. The values written to the arrays are read back out at a later time and a check-sum is performed. The results of the check-sum are cumulative and shown to the outside world through the FAILURE pin.

The algorithm is as follows:

# Initialization
Disable all interrupts
Tristate the external Bus

# Self-Test
Assert the FAILURE pin

Loop:
- Write the IEU RAM array
- Write the IFU CACHE
- Write the TLB CACHE (80960MC only)
- Read the IEU RAM array and Literals
- Perform a checksum update
- Read the IFU CACHE
- Read the TLB Cache (80960MC only)
- Perform a checksum update
- Repeat the loop until all locations have been tested

Read the MIS ROM until done

Perform a checksum update
Check Results

IF checksum is wrong Assert the FAILURE pin and stop
ELSE Deassert the FAILURE pin

Assert the FAILURE pin

Read the first 8 words from memory
Perform a checksum update
Check Results

IF checksum is wrong Assert the FAILURE pin and stop
ELSE Deassert the FAILURE pin
# End Self-Test

# Finish Initialization
Fetch first instructions from initialization IP
Enable Instruction Decoding

Notice in the above algorithm that the FAILURE pin is actually asserted, then deasserted upon successful completion of the self-test.

This algorithm does not trace out the entire initialization procedure followed upon power-up and reset. Refer to the chapter “Processor Management and Initialization” in the “80960KB Programmer’s Reference Manual” for more complete information on the initialization sequence and the “80960KB Data Sheet” for the proper reset sequence.

CONCLUSION

The automatic self-test integrated into the 80960Kx and 80960MC greatly enhance the diagnostic effort in an embedded system by ensuring the embedded processor is working properly before any system diagnostics are carried out. This extra step in diagnostics along with the comprehensive testing performed on each and every part shipped significantly reduces the possibility of a defective part being allowed to operate in an embedded system.
Intel’s internally developed 80960 architecture allows embedded system designers to take advantage of silicon technology advancements without architectural limits. The 80960, developed from scratch for embedded control applications, eliminates architectural obstacles to state-of-the-art implementation techniques that allow parallel and out-of-order instruction execution.

In introducing the 80960 architecture, I distinguish between the architecture and the microarchitecture of an implementation. A microarchitecture is an actual implementation of the architecture's instruction set and programming resources. Different microarchitectures may have different pipeline construction, internal bus widths, register set porting, cache parameterization (two-way, four-way, and so on), and degrees of parallelism, or may not execute instructions out of order. The architecture is specified in such a way that wide latitude is available for future microarchitectural advancements. In this way both very high performance and highly integrated controllers can be built using the 80960 architecture.

Principally, the 80960 architecture allows, for at least the next decade, silicon technology and microarchitectural advances to translate directly into increased performance without architectural limitations. While the common performance target of typical RISC architectures is an execution speed of one instruction per processor clock cycle, the 80960 architecture targets the execution of multiple instructions per clock cycle (fractional clock cycles per instruction). By defining an architecture that supports parallel instruction execution and out-of-order instruction execution, we do not constrain performance advances to the system clock cycle.

Additionally, the 80960 has been optimized for the wide range of applications that are unencumbered by a need to be compatible with an existing embedded control architecture. These applications are often very cost sensitive, require a different mix of instructions than reprogrammable applications, have demanding interrupt response requirements, and use real-time executives rather than full-blown operating systems. With these factors in mind, we developed the 80960 while avoiding architectural constructs that would restrict an implementation's capability to execute multiple instructions in one clock cycle.

David P. Ryan
Intel Corporation

June 1988
The architecture also allows application-specific extensions such as:

- instruction set extensions (floating-point operations),
- special registers,
- larger caches,
- multiple caches,
- on-chip program and data memory,
- a memory management and protection unit,
- fault-tolerance support,
- multiprocessing support, and
- real-time peripherals (DMA, analog/digital, serial ports).

The 80960's instruction set has also been optimized for embedded control applications. It offers Boolean operations more powerful than those of the 8051 family. Single instructions access frequently executed functions for increased code density and performance. They include CALL, RET, Compare_and_Branch, Conditional_Compare, Compare_and_Increment or Decrement, and Bit Field Extract.

A priority interrupt structure simplifies the management of real-time events, and, along with a user/supervisor model, supports fast, safe, real-time kernel operation. A generalized fault-handling mechanism simplifies the task of detecting errant arithmetic calculations or other conditions that typically require a significant amount of user code runtime support. The 80960 does not require sophisticated, optimizing high-level-language compilers to achieve high performance. However, no obstacles to performance enhancements via a good optimizing compiler exist.

Since products based on the 80960 have high performance levels, even without code optimization, many users will attain their required system performance with 80960 products without having to understand the parallelism of the implementation they are using.

### Architecture overview

The 80960 can best be described as a register-rich, load/store architecture with an instruction set designed to let implementations exploit pipelining and parallel execution strategies. Direct embedded control support includes:

- an optimized instruction set,
- a flexible interrupt structure,
- a user-supervisor model,
- a powerful fault-handling structure, and
- multiprocessing hooks and debug support.

The architecture extends easily.

Figure 1 shows a logical block diagram of the architecture's programming resources. The 32-bit memory space is flat. Data moves between memory and registers via load and store instructions. Processing elements surrounding the registers manipulate parallel data; they receive their instructions from the
instruction sequencer.

The instruction sequencer reads multiple instructions simultaneously from an instruction cache and presents the instructions in parallel to the processing elements as appropriate. When the instruction stream or an asynchronous event requires a context switch, the local variables from the suspended procedure move to the register cache. Memory accesses to save previously suspended register sets occur only when the register cache is filled. The implementation determines the number of architecturally transparent register sets that can be cached.

We expect different implementations of the processing elements in an 80960-based controller—optimized for specific applications—will evolve. We defined a standard core architecture to maintain binary-compatible instruction sets across all implementations for leveraging compiler and real-time kernel development. Subsequent references to the 80960, without an alpha suffix, refer to the architecture. References to an 80960XY pertain to actual implementations of the core architecture, which may contain architectural extensions and/or on-chip peripherals. (See the accompanying box for a discussion of three implementations.)

Implementations of the 80960

As an example of an actual implementation of the core architecture, consider the first 80960 implementation, the 80960KB controller. The KB provides architectural extensions such as floating-point operations (Figure A). Its on-chip floating-point unit implements the IEEE floating-point standard, including transcendental support. Floating-point performance exceeds 4 million Whetstone operations per second at 20 MHz. The 80960KB integrates a 512-byte instruction cache and an interrupt controller on chip.

Another member of the family, the 80960KA controller, fits the KB socket but lacks the on-chip floating-point unit. The 80960MC controller, a military-qualified version similar to the KB, adds Ada tasking support and a memory management and protection unit.

The 80960KA, KB, and MC microarchitecture sustains execution of up to one instruction per clock cycle at 20 MHz (20 native MIPS). It performs a variety of benchmark programs seven to 10 times faster than a VAX 11/780 (7 to 10 VAX MIPS).
Register model

The user directly accesses thirty-two 32-bit general-purpose registers and 32 special-function registers (SFRs). (Refer to Figure 1.) Of the 32 general registers, 16 are global registers and 16 are local registers. Data in the 16 global registers remain visible and unchanged when crossing procedure boundaries, characteristics exhibited by "normal" registers in other architectures. The CALL instruction caches the local registers and the RET instruction restores them. The SFRs provide a real-time register interface to on-chip peripherals. The SFRs, the contents of which are not defined by the architecture, control implementation-specific hardware.

When a procedure call occurs, data in the local registers automatically move to a register cache. Thus, the called procedure is not required to explicitly save the local registers. When the called procedure executes a return instruction, the data in the local registers prior to the call are restored. The call/return sequence does not affect the global registers, which can be used to pass parameters and results between procedures.

A large, general-purpose register set greatly reduces the number of memory requests required to perform a task. Various optimization techniques can use large-register sets to remove data dependencies that would otherwise prohibit parallel instruction execution. For example, a hypothetical implementation of the architecture could provide parallel execution of two ADD instructions. Having many general-purpose registers with which to work simplifies code optimization so that neither ADD instruction references a source that was the destination of the other ADD instruction. Under such circumstances, two ADD instructions per cycle could be sustained.

Note that such program optimizations are not required. Any program using the architecture's core instruction set and not referencing the SFR address space or external I/O, whether optimized or not, will function correctly on any implementation without modification. Even if the optimization rules are radically different between implementations, code that is optimized for one implementation will run correctly on another implementation without modification or recompilation.

The architecture also guarantees that data dependencies will be correctly handled without programmer intervention. For example, execution of an arithmetic instruction will be delayed if it uses a register that is waiting for data to be loaded from memory. However, other instructions that do not use the register in question could be executed immediately with all data dependencies automatically maintained. This capability is only beneficial when a large register set is present to remove avoidable data dependencies.
Core instruction set

The 80960 instruction set is similar in design to engines in reduced instruction set computers. Because the 80960 was designed to avoid performance bottlenecks resulting from instruction decoding times, all opcodes are 32 bits (one word) in length and must be aligned on word boundaries. The only two-word (64-bit) instruction format loads 32-bit immediate constants and assists effective address calculations. Generally, load, store, and control instructions access memory. All other instructions access only registers.

Thirty-two-bit opcodes provide tremendous flexibility in instruction encoding. However, performance penalties associated with code size can occur when often-used complex instruction sequences are not available in a one-word format. Large code size not only increases system cost due to larger memory requirements but also results in penalties in cache utilization and bus-bandwidth requirements.

The 80960 includes one-word multifunction instructions to avoid such code density problems that increase memory requirements and to allow complex functions to be parallelized. For example, the CALL and RET instructions provide one-word encodings for sequences that otherwise require several instructions. Implementations of the architecture could perform the CALL/RET operations in parallel with other instruction execution. Or, the processor could execute from an on-chip ROM a similar sequence of simple 80960 instructions that the user would have to write if CALL/RET instructions were not in the architecture. Executing the code from permanent on-chip storage results in higher performance than does requiring the instructions to be fetched and cached every time they are used. In addition to ensuring higher performance, or possibly parallel performance, implementations of such functions as ROMed assembly code sequences—triggered by a one-word opcode—are more silicon efficient than are increases of on-chip cache sizes to deal with poor code density. For example, a ROM cell is typically the fourth size of a RAM cell.

The availability of complex instructions in the architecture does not prohibit the programmer from bypassing them if simpler functions are desired. For example, a BAL (Branch_and_Link) instruction can be used to call a procedure that does not require a new set of local registers. The BAL instruction saves the next instruction pointer in a register and branches to the specified destination. When the procedure is ready to return, it executes an indirect branch to the return instruction pointer that was saved. It is likely that BAL will always be faster than CALL since no local registers are being saved. The programmer can use whichever method best suits the circumstances.

Figure 2 shows the 80960 instruction encodings. Most instructions appear in the REG format, which specifies an opcode and three registers/literals (one of 32 registers, or a constant value in the range 0 to 31).

The COBR format specifies a set of compare and branch instructions. The CRTL format covers branch and call instructions. The MEM formats support load and store instructions.

Much of the instruction set is what one would expect to encounter in all processors (ADD, MUL, SHIFT, BRANCH); however, some instructions deserve special mention.

- The register-register move instructions transfer one, two, three, or four register values. The same is true for the load and store instructions (for example, LDQ loads four words into four registers).
- In addition to the normal shift instructions, the SHRDI instruction provides an adjustment to the result so the instruction can be used to divide a value by a power of two. (Normal right-shift instructions do not divide correctly when the value is negative and odd.)
- All logical operations of two operands are provided (AND, NOTAND, ANDNOT, NOR).
- An extensive set of bit instructions exists (SET BIT, CLEAR BIT, NOT BIT, SCAN a register for the first 0, or 1, BRANCH on bits set or clear), as well as instructions for accessing bit fields (MODIFY, EXTRACT).
- Single-instruction COMPARE_AND_BRANCH encodings optimize code density for these frequently executed operations.
- Conditional compare (CONCOMP) instructions speed bounds checking.

Table 1 on the next page summarizes the 80960 core architecture instruction set.

The architecture directly supports integer (signed) and ordinal (unsigned) data types. Bits, bit fields, bytes, short words, words, and double words can be manipulated in registers and transferred to and from memory. Triple words and quad words can also move between the registers and memory.

Register operations

Most 80960 instructions operate on registers. The architecture provides arithmetic, logical, bit and fit field, data movement, and comparison operations. To take full advantage of the large register set, three-operand instructions specify any register as one or both sources and/or the destination of an operation.

Arithmetic and logical. The architecture supports both standard and extended arithmetic operations. Add, subtract, multiply, and divide operations are available on 32-bit integers and ordinals. Extended multiply operates on two 32-bit ordinals and generates a 64-bit result. Extended divide divides a 64-bit ordinal by a 32-bit ordinal, producing a 32-bit quotient and 32-bit remainder. Addition and subtraction with carry allow 32-bit ordinals to provide extended precision adds and subtracts.
## 80960 architecture

### Table 1. 80960 instruction summary.

<table>
<thead>
<tr>
<th>REGISTER OPERATIONS</th>
<th>CONTROL OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARITHMETICS</strong></td>
<td><strong>BRANCH</strong></td>
</tr>
<tr>
<td>`add[i</td>
<td>o]` Add</td>
</tr>
<tr>
<td><code>addc</code> Add with Carry</td>
<td><code>bx</code> Branch Extended (32-Bit Indirect Branch)</td>
</tr>
<tr>
<td>`sub[i</td>
<td>o]` Subtract</td>
</tr>
<tr>
<td><code>subc</code> Subtract with Borrow</td>
<td><code>b[*]</code> Branch on Condition</td>
</tr>
<tr>
<td>`mul[i</td>
<td>o]` Multiply</td>
</tr>
<tr>
<td><code>emul</code> Extended Multiply</td>
<td><code>cmpob[*]</code> Compare Ordinal and Branch on Condition</td>
</tr>
<tr>
<td>`div[i</td>
<td>o]` Divide</td>
</tr>
<tr>
<td><code>divc</code> Extended Divide</td>
<td><code>fault[*]</code> Fault on Condition</td>
</tr>
<tr>
<td>`rem[i</td>
<td>o]` Remainder</td>
</tr>
<tr>
<td><code>modi</code> Modulo Integer</td>
<td></td>
</tr>
<tr>
<td>`sh[lo</td>
<td>ro]</td>
</tr>
<tr>
<td><strong>MOVEMENT</strong></td>
<td><strong>PROCEDURE</strong></td>
</tr>
<tr>
<td>`mov[l]</td>
<td>q]` Movie registers to registers</td>
</tr>
<tr>
<td><code>lda</code> Load Address</td>
<td><code>callx</code> Call Extended (32-Bit Indirect Call)</td>
</tr>
<tr>
<td><strong>COMPARISON</strong></td>
<td><code>calls</code> System Procedure Call</td>
</tr>
<tr>
<td>`cmp[l]</td>
<td>o]` Compare</td>
</tr>
<tr>
<td>`cmpdec[l]</td>
<td>o]` Compare and Decrement</td>
</tr>
<tr>
<td>`cmpinc[l]</td>
<td>o]` Compare and Increment</td>
</tr>
<tr>
<td>`cmpeq[l]</td>
<td>o]` Conditional Compare</td>
</tr>
<tr>
<td><code>test[*]</code> Test for condition</td>
<td></td>
</tr>
<tr>
<td><code>scanbyte</code> Scan for matching byte</td>
<td></td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td><strong>ENVIRONMENT</strong></td>
</tr>
<tr>
<td><code>and</code> dst := src1 &amp; src2</td>
<td><code>modpc</code> Modify Process Controls</td>
</tr>
<tr>
<td><code>andnot</code> dst := src2 &amp; (~src1)</td>
<td><code>modac</code> Modify Arithmetic Controls</td>
</tr>
<tr>
<td><code>notand</code> dst := (~src2) &amp; src1</td>
<td><code>modtc</code> Modify Trace Controls</td>
</tr>
<tr>
<td><code>nand</code> dst := ~(src2 &amp; src1)</td>
<td><code>flushregs</code> Flush Local Register Cache to Memory</td>
</tr>
<tr>
<td><code>or</code> dst := src1</td>
<td>src2</td>
</tr>
<tr>
<td><code>nor</code> dst := ~(src2</td>
<td>src1)</td>
</tr>
<tr>
<td><code>ornot</code> dst := src2</td>
<td>(~src1)</td>
</tr>
<tr>
<td><code>notor</code> dst := (~src2)</td>
<td>src1</td>
</tr>
<tr>
<td><code>xor</code> dst := (src2</td>
<td>src1) &amp; (~src2 &amp; src1)</td>
</tr>
<tr>
<td><code>not</code> dst := ~src</td>
<td></td>
</tr>
<tr>
<td><code>rotate</code> Rotate Bits</td>
<td></td>
</tr>
<tr>
<td><strong>BIT AND BIT FIELD</strong></td>
<td><strong>DEBUG</strong></td>
</tr>
<tr>
<td><code>setbit</code> Set a Bit</td>
<td><code>mark</code> Conditionally generate Trace Fault</td>
</tr>
<tr>
<td><code>clrbit</code> Clear a Bit</td>
<td><code>fmark</code> Unconditionally generate Trace Fault</td>
</tr>
<tr>
<td><code>notbit</code> Toggle (invert) a Bit</td>
<td></td>
</tr>
<tr>
<td><code>chkbit</code> Check a Bit and set condition code</td>
<td></td>
</tr>
<tr>
<td><code>alterbit</code> Change a Bit according to an operand</td>
<td></td>
</tr>
<tr>
<td><code>scanbit</code> Search src for most significant set bit</td>
<td></td>
</tr>
<tr>
<td><code>spanbit</code> Search src for most significant cleared bit</td>
<td></td>
</tr>
<tr>
<td><code>extract</code> Extract specified bit pattern from a word</td>
<td></td>
</tr>
<tr>
<td><code>modify</code> Modify selected bits in dst with src</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
& i = \text{integer}, \ a = \text{ordinal}, \ b = \text{byte}, \ s = \text{short}, \ w = \text{word (32-bits)}, \ l = \text{long}, \ t = \text{triple}, \ q = \text{quad}, \\
& \text{lo} = \text{left ordinal}, \ li = \text{left integer}, \ ro = \text{right ordinal}, \ ri = \text{right integer}, \ di = \text{right dividing integer}, \\
& \text{dst} = \text{destination}, \ src = \text{source}, \ x = \text{extended}, \\
& ^* = \text{Conditions: if [equal | not equal | less | less or equal | greater | greater or equal | ordered | unordered ]}
\end{align*}
\]
Arithmetic shift operations support 32-bit ordinals. Logical shift instructions operate on 32-bit integers, and a 32-bit register value can also be rotated. In addition, all possible two-operand, bitwise Boolean operations exist: AND, NOTAND, ANDNOT, OR, NOR, XNOR, NOT, NOTOR, ORNOT, and NAND.

Bit and bit field. Bit operations allow bits in the registers to be set, cleared, toggled, and moved to or from the condition codes. SCAN and SPAN operations provide ways to find the most significant set or cleared bit in a register.

The 80960 contains two bit field instructions, EXTRACT and MODIFY. The EXTRACT instruction shifts a bit field in a register to the right and fills in the bits to the left of the bit field with zeros. The MODIFY instruction moves a specified bit field from one register to another when no adjustment change in bit position is required.

Data movement. A set of data movement (MOV) instructions allows register values to be copied to other registers. The MOV instructions can move from one to four registers at once. The load and store operations described later move data to and from memory.

Comparison. These instructions compare operands and set the resulting condition codes in the arithmetic controls register (Figure 3). The 80960’s condition codes listed in Table 2 provide the arithmetic flag function of other architectures, in a way that allows maximum parallel execution.

In general, only compare instructions set the 80960’s condition codes and conditional instructions use them. To perform an ADD followed by a conditional branch when the result is zero, a Compare_and_Branch instruction must be executed after the ADD because arithmetic instructions do not alter the condition codes.

Although not generally noticed in a sequential execution environment, a parallel environment mandates the decoupling of the condition codes from the instruction set. The 80960 allows multiple instructions to execute simultaneously, thus giving ambiguous meaning to a set of condition codes that are altered by multiple arithmetic instructions in the same clock cycle. The 80960 approach separates condition checking and decision making from all other instructions to provide flexibility in reordering instructions for parallel execution.

The 80960 compare instructions compare both integers and ordinals. A subset of the compare instructions increments or decrements an operand after the comparison.

Memory operations

The load/store nature of the architecture decouples memory references from instruction execution. Register set and memory instructions can be executed simultaneously. Since the load data may take some time to
arrive at the CPU, the load requests can be advanced in the instruction stream to overlap memory access time with other data-independent CPU operations.

**Load/store.** The load and store instructions copy bytes, short words, words, or multiple words to or from memory and registers. When a load integer is specified for an 8-bit or 16-bit quantity, the CPU extends the data’s sign to fill 32 bits before writing the destination register. When a load ordinal is specified for an 8-bit or 16-bit quantity, the CPU attaches leading zeros to the data to fill 32 bits before writing the destination register. The store instructions allow the destination data width to be a byte, short word, word, or multiple words. When a store byte, or short word is performed, the CPU automatically formats the data being written according to the data type (integer or ordinal).

**Addressing modes.** The architecture supports 11 addressing modes for memory operations, as summarized in Table 3. The addressing modes selected for support provide a broad range of most-often-used simple modes. We chose a rich set of addressing modes to allow optimization for code density as well as speed.

**Literals** are immediate 5-bit numbers that can range from 0 to 31. Literals may be used as operands in any register operation.

The **Register address mode** is used when an operand specifies a register number (g0, r5).

The **Absolute Offset address mode** specifies the absolute address of the target as an offset from the current instruction pointer. The offset is encoded in the memory instruction opcode. If the offset is outside the range of 0 to 2048, the assembler generates a two-word instruction in which the second word is a 32-bit displacement.

**Register Indirect addressing** allows the address of the target to be specified by the contents of a register. An immediate offset or displacement can be added to the register to form the effective address. An index (scaled by 2, 4, 8, or 16) may also be added.

Memory operations can also specify target addresses relative to the instruction pointer, a capability useful in creating relocatable data and code.

**Atomic memory operations.** Two atomic memory operations support multiprocessing environments with more than one processor accessing the same memory, atomic add (ATADD) and atomic modify (ATMOD). The ATADD instruction causes an operand to be added to the value in the specified memory location. The ATMOD causes bits in the specified memory location to be modified under control of a mask. These instructions perform memory-to-memory, read-modify-write operations with a locked bus to prevent data corruption.

**Control operations**

Control operations include those instructions that could result in the redirection of program flow. CALL, RET, BRANCH, and COMPARE_AND_BRANCH instructions fall into this category.

**Procedure calls.** The CALL instruction causes the local registers to be preserved and redirects program flow to a point indicated by an offset encoded in the instruction. The Call Extended (CALLX) instruction dif-

---

### Table 3. Addressing modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Assembler Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>value</td>
<td>0x12</td>
</tr>
<tr>
<td>Register</td>
<td>register</td>
<td>r6</td>
</tr>
<tr>
<td>Absolute offset</td>
<td>offset</td>
<td>Label + 3</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>abase</td>
<td>(r6)</td>
</tr>
<tr>
<td>Register Indirect with offset</td>
<td>abase + offset</td>
<td>Label + 3 (r6)</td>
</tr>
<tr>
<td>Register Indirect with index</td>
<td>abase + (index • scale)</td>
<td>(r6) [7 * 4]</td>
</tr>
<tr>
<td>Register Indirect with index and displacement</td>
<td>abase + (index • scale) + displacement</td>
<td>Label + 3 (r6) [7 * 4]</td>
</tr>
<tr>
<td>Index with displacement</td>
<td>(index • scale) + displacement</td>
<td>Label [r6 * 4]</td>
</tr>
<tr>
<td>IP with displacement</td>
<td>IP + displacement + 8</td>
<td>Label (IP)</td>
</tr>
</tbody>
</table>
fers in that it allows a 32-bit value to provide the CALL destination. The destination can either be encoded in the instruction or specified by a register value (for example, indirect call). The Call System (CALLS) instruction gets its target address from a table of system procedure addresses explained later. The Return (RET) instruction returns control to the calling procedure and restores the local registers of the calling procedure.

The semantics of the CALL/RET allow an optimization known as register caching. A register cache keeps the context (local registers) of the most recently executing subroutines on chip so that CALL/RET instructions do not have to access memory to save or restore the local registers.

When a CALL instruction executes, the 80960 allocates a new set of 16 local registers from a pool of register sets for the called procedure. If the pool is depleted, a new register set is allocated by taking one associated with an earlier procedure and saving it in memory. A RET instruction causes the most recently cached local register set to be restored, freeing a register cache location.

The register cache contributes to performance in four ways:

- It significantly reduces the saving and restoring of registers that are usually performed when crossing subroutine boundaries.
- Since the local register sets are mapped into the stack frames, the linkage information that normally appears in stack frames (pointer to previous frame, saved instruction pointer) is contained in the local registers. Most call and return instructions execute without causing any references to off-chip memory.
- It allows compilers to map most or all of a procedure's local variables directly into registers.
- It provides a large number of registers (16 local and 16 global), which can be exploited by optimizing compilers.

The procedure stack (see Figure 4) reserves space for the cached registers of suspended procedures. When a register set must be flushed from the register cache to memory, it moves to the reserved stack frame space.

When a new procedure is entered, the 80960 allocates space for the procedure's register set as the only contents of its stack frame, although no memory accesses will occur unless the register cache is full. If the procedure desires more space on the stack for autovariables or parameter passing, it adjusts the stack pointer to reserve as much space as it needs. The procedure can then access this space using stack pointer relative addressing so long as the procedure is active. When the procedure returns, its stack is automatically reclaimed.

Branch_and_Link (BAL) performs a procedure call without saving the local registers. The 80960 saves the return instruction pointer in a global register and redirects program flow. To return from a routine that is invoked by a BAL, a BX (Branch Extended) is performed. BAL allows fast subroutine calls to leaf procedures without allocating (and possibly displacing) a new register set. Since a leaf procedure calls no other procedure, its registers can be allocated out of those remaining in the current set.
Branching. Advanced architectures have yet to deal cleanly with the dreaded branch, although some existing methods try and minimize the instruction pipeline breaks caused by branches and conditional branches. One method used by other architectures is a delayed branch. This method requires that a valid instruction always be placed after every branch. The delayed branch mechanism exposes the pipeline to the programmer and makes it easy to write code that “breaks.” Compilers also have a difficult time finding useful instructions to always fill the blank pipeline slots following a branch and insert NOPs about 30 percent of the time. Furthermore, in architectures with a delayed branch mechanism, microarchitectures will be constrained in their enhancement choices.

The 80960 alternative to a delayed branch hides the pipeline and microarchitecture implementation from the programmer and allows transparent performance enhancements. For example, an 80960 microarchitecture that detects branches ahead of the executing code could fetch the branch destination to keep the pipeline full. In essence, “branch lookahead” allows branches to be executed in zero clock cycles.

Branches can be unconditional or conditional. The Branch and Branch Extended instructions perform unconditional redirection of program flow without linkage. Branch and Branch Extended differ in the width of the target address offset provided. The Branch instruction includes an encoded offset in the one-word instruction (MEMA format), whereas Branch Extended branches to the location pointed to by a register or an encoded 32-bit displacement (MEMB format).

The conditional branches use the condition codes in the arithmetic controls register to determine whether or not to take the branch. The 80960 provides all combinations of branch conditions.

Branch lookahead works well with unconditional branches but would be of marginal benefit on conditional branches since the branch target, or the instruction after the branch, cannot be executed until after evaluation of the branch condition. Pipeline breaks would, therefore, be inevitable even if the microarchitecture implemented some sort of hardware prediction mechanism. To reduce the effect of the conditional branch on performance, the 80960 defines two types of conditional branches, those that are usually taken and those that aren’t usually taken. The implementation can then guess which way the branch is going to go, based upon an excellent resource capable of prediction—the programmer. Only in the case of a programmer’s wrong prediction would a pipeline stall occur. Furthermore, compilers will take advantage of branch prediction when they detect loops.

It is either obvious, or uncertain, at the time the program is written which way the branches will branch most often during execution. If the likely branch outcome is obvious, the type of branch to use will be obvious. In the cases where runtime factors determine the branch path, the branch types can be selected to reduce the time through the longest path or to reduce the average path time.

Compare and branch. The compare and branch instructions support integers and ordinals. The CPU compares two operands; the result determines the branch taken. This frequently used operation is one instruction that increases performance and improves code density. The 80960 provides all combinations of branch conditions, in addition to branch-on-bit instructions.

Instruction cache

As processors increase in speed, the traffic between processor and memory becomes a significant performance bottleneck. To effectively reduce this bottleneck, we incorporated an instruction cache within the processor.

An on-chip instruction cache is highly desirable for two reasons. Caching instructions on chip greatly reduces system bus loading and the criticality of the system’s memory access speed in a parallel execution environment. However, an instruction cache plays an additional role. The only way to cause multiple instructions to execute simultaneously is to decode multiple instructions simultaneously. An on-chip instruction cache gives instruction decode the capability of looking downstream and decoding and dispatching multiple instructions simultaneously for parallel execution.

The advantage of an instruction cache over a prefetch queue, a technique used in most high-performance microprocessors to date, is that a queue does not reduce the memory traffic for instructions. It only attempts to distribute the traffic more efficiently. A cache’s most obvious effect occurs with execution loops, common in embedded control applications. After a loop is first executed, successive iterations of the loop generate no memory references for instruction fetches. Likewise, when a small, low-level procedure concludes and executes a RET instruction, the cache for the high-level routine to which it is returning likely still resides in the cache. Thus, we reduce the sensitivity of instruction execution speed to slow memory and free valuable bus bandwidth for other operations.

Having an instruction cache requires special considerations in applications that employ self-modifying code or uploadable programs. In general, embedded applications are unaffected. However, for 80960 chips targeted at embedded applications in which volatile code exists, we will provide implementation-specific cache features. For example, implementations could provide a bus input pin that prohibits the data being read from being cached, a method for flushing the cache, a transparent instruction cache, a cache disable bit, or some other feature tuned to the application.
To allow implementations of the 80960 latitude in the amount and type of cache provided, the architecture does not specify the instruction cache parameterization.

User-supervisor protection

The architecture provides a mode and stack switching mechanism called the user-supervisor protection model. This protection model allows a system design in which the kernel code and data reside in the same address space as the user code and data. However, the access to the kernel procedures (called supervisor procedures) occurs through a specified interface. A data structure called the System Procedure Table provides this interface (Figure 5).

The 80960 references the System Procedure Table when a System Call (CALLS) instruction executes. This call is similar to a local call, except that the processor gets the location of the called procedure from the System Procedure Table. Figure 6 illustrates the use of the CALLS instruction. CALLS requires a procedure-number operand that is used as an index into the table.

The System Procedure Table entry referenced by CALLS specifies an entry pointer and an entry type for the called procedure. The 80960 invokes two types of system procedures, local and supervisor. A procedure that is specified as a local procedure is invoked as if it were called by the CALL or CALLX instructions, except that the processor gets the entry point of the called procedure from the System Procedure Table. Referencing a supervisor procedure, on the other hand, switches the processor to the supervisor execution mode and to the supervisor stack.

Real-time kernel procedures in the supervisor mode execute using a different stack than the one used to execute application programs procedures. Special, supervisor-only instructions also execute in supervisor mode. The MODPC instruction (used to change the processor priority) is always a supervisor instruction. Instruction set extensions that control on-chip hardware are also likely to be restricted to the supervisor mode.
The processor remains in the supervisor mode until
the procedure that caused the original mode switch per-
forms a return. Switching stacks and protecting against
stack corruption help maintain the integrity of the
kernel. For example, the mechanism allows access to
system debugging software or a system monitor even if
the application crashes.

Interrupts

The 80960 architecture

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Pending priorities</th>
<th>0 - 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pending interrupts</td>
<td>4 - 35</td>
</tr>
<tr>
<td></td>
<td>Entry 8</td>
<td>36 - 39</td>
</tr>
<tr>
<td></td>
<td>Entry 9</td>
<td>40 - 43</td>
</tr>
<tr>
<td></td>
<td>Entry 10</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Entry 243</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>979</td>
</tr>
<tr>
<td></td>
<td>Entry 252</td>
<td>980 - 1011</td>
</tr>
<tr>
<td></td>
<td>Entry 255</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1027</td>
</tr>
</tbody>
</table>

The interrupt table seen in Figure 7 holds the 32-bit
pending priorities field, the 256-bit pending interrupt
field, and the 248 interrupt vectors. Within each pro-
cessor priority the 80960 contains eight vectors, eight
associated pending interrupt bits, and one pending
priority bit. The pending priorities field indicates the
priorities at which pending interrupts await. The pen-
ding interrupt field indicates exactly which requested
interrupts have not yet been serviced.

A pending priority bit is simply the OR of all eight
pending interrupt bits at a particular priority. This field
optimizes checking for pending interrupts by the pro-
cessor. When an interrupt request will not be serviced
immediately, the 80960 sets the bit in the pending inter-
rupt field associated with the request. It also sets the
pending priorities bit associated with the priority of the
request. When the running priority of the processor
drops below that of the pending interrupt, the 80960
services the interrupt and clears the associated pending
bit. The CPU also clears the associated pending priority
bit if appropriate.

Faults

Processors use fault mechanisms to handle exceptions
or errant conditions that a program may or may not be
capable of correcting. We defined the 80960’s
fault mechanism for an environment in which parallel
or out-of-order execution occurs. When a fault is
generated, the processor calls the appropriate fault
handler. The 80960 automatically provides the handler
with an extensive set of information about the faulting
condition for correction or analysis.

It is possible that when a fault is detected not enough
information would exist to determine the exact instruc-
tion that faulted. For example, when multiple instruc-
tions execute in one clock cycle, multiple faults could
correct in a single clock cycle. This “imprecise” condition
could generate a fault that we call imprecise. A tightly
coupled fault handler may be able to recover proper
program execution when an imprecise fault occurs.
Precise faults are those from which recovery is easy.

The 80960 provides two controls over the generation
of imprecise conditions and faults. The first fault con-
control method, a global control bit, puts the processor in
a mode where no imprecise conditions are created (No
Imprecise Faults, or NIF mode). In this mode, the
80960 restricts parallel execution. All faults are precise.
The NIF bit can be used to create a critical region in
which all faults are precise. The second fault control
mechanism is the Synchronize__Faults (SYNCF)
instruction. This instruction halts execution until all
pending operations complete, and all faults up to that
point have been signaled. It is useful on Ada block
boundaries where different blocks can have different
fault handlers.

An 80960 implementation detects various conditions
in code or in its internal state (called fault conditions)
that could cause the processor to deliver incorrect or in-
appropriate results, or that could cause it to take an
undesirable control path. For example, the 80960 can
recognize (if enabled by the user) divide-by-zero and
overflow conditions on integer calculations as a fault.
The architecture also recognizes inappropriate operand
values and attempts to execute unimplemented op-
codes, among other conditions, as faults.

When a fault is detected, the system processes it im-
mediately and independently of the program or han-
der that is executing at the time. The fault mechanism
is similar to that used by the interrupts. Several fault
types exist, in which the fault type determines which entry in the Fault Table (Figure 8) is invoked for a particular fault. The Fault Table contains one entry for each fault type. The entry defines a particular fault handler routine as a local procedure or a system procedure. When the fault handler is a local procedure, the Fault Table entry contains the address of the procedure entry point. When the fault handler is a system procedure, the Fault Table entry contains the system procedure number, which selects the correct entry point from the System Procedure Table described earlier.

Figure 9 describes the fault record, which is the information provided to a fault handler when a fault occurs. Table 4 on page 76 summarizes the fault types and subtypes that are currently defined in the 80960 architecture. As extensions to the architecture that consume additional fault types become available, the encoding of fault types and subtypes will occur in such a way that every implementation capable of recognizing similar faulting conditions encodes them identically. For example, the 80960KB adds the floating-point faults (fault type 4). Any other 80960 implementations that also recognize floating-point faults also encode them as fault type 4.

**Debug support**

Another objective of the architecture is to support software debugging and tracing. A trace-controls register enables most of this support. The trace controls detect any combination of the following events:

- Instruction execution (single step),
- Execution of a Taken Branch instruction,
- Execution of a Call instruction,
- Execution of a Return instruction,

![Figure 8. Structure of the fault table (a); an entry to reference a local procedure (b); and an entry to reference a system procedure (c).](image)

![Figure 9. Fault record information. The return pointer r2 is also provided.](image)
80960 architecture

Table 4. Fault types and subtypes.

<table>
<thead>
<tr>
<th>Fault type</th>
<th>Fault Subtypes</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Overflow, underflow</td>
<td>Integer overflows/ divide by zero</td>
</tr>
<tr>
<td>Constraint</td>
<td>Range</td>
<td>If FAULT_IP taken</td>
</tr>
<tr>
<td>Protection</td>
<td>Length</td>
<td>Procedure # in CALLS out of range</td>
</tr>
<tr>
<td>Machine</td>
<td>Bad access</td>
<td>Memory access failed to complete</td>
</tr>
<tr>
<td>Type</td>
<td>Mismatch</td>
<td>Tried to execute supervisor instruction in nonsupervisor mode</td>
</tr>
<tr>
<td>Operation</td>
<td>Invalid opcode, invalid operand</td>
<td>Tried to execute invalid opcode, or an operand in a valid opcode was invalid</td>
</tr>
<tr>
<td>Trace</td>
<td>Instruction, branch, call, return, prereturn, supervisor, breakpoint</td>
<td>Trace event occurred</td>
</tr>
</tbody>
</table>

- Detection that the next instruction is a Return instruction,
- Execution of a supervisor or system call, and
- Breakpoint (hardware breakpoint or execution of a breakpoint instruction).

When a trace event is detected, the processor generates a trace fault to give control to a software debugger or monitor. Since all 80960 implementations are likely to have an on-chip cache, external hardware cannot trace the flow of instruction execution by monitoring the external bus. Therefore, to trace instruction execution, a debugger could enable the BRANCH, CALL, and RET trace faults and reconstruct the instruction-by-instruction flow of a program. This method, however, will not provide transparent, or real-time tracing. When noninvasive emulation is desirable, the user should employ an in-circuit emulator.

The 80960, an extensible embedded control architecture, maximizes computational and data processing speed through parallel execution. The first implementation of the architecture (80960KB) achieves single-clock execution of instructions, while fractional clock instruction rates are architecturally unhindered and will be available in future implementations.

Acknowledgments

Too many people contributed to the 80960 effort to list them here. However, I relied upon the following, either in person or through their writings, in developing this article: Dave Budde, Glen Hinton, Mike Imel, Konrad Lai, Glenford J. Myers, Lew Pacely, Fred Pollack, Rob Riches, Frank Smith, and Randy Steck.

Bibliography


David P. Ryan, a senior applications engineer in Intel's embedded controller operation, supervises the Arizona-based 32-bit applications team. He began work on 32-bit embedded controller requirements in early 1985 and full-time work on 32-bit products in early 1987. Before this, he supported the MCS-96 family of 16-bit microcontrollers.

Ryan holds a BSEE from Arizona State University and an MBA from Pennsylvania State University.

Embedded Controllers
Push Printer Performance

by Phillip Bride,
Intel Corp.,
Hillsboro, OR

Touting speed and the ability to address large amounts of inexpensive memory, on-board floating-point calculations, a large register set, and an on-board interrupt controller, the 80960KB 32-bit processor promises to pump up laser printers.

Breaking the performance bottleneck in laser printers calls for a controller with plenty of muscle. To see how much muscle, look at what a high-range laser printer must do: mix text and graphics; resolve at least 300 pixels, or dots, per inch; deliver at least 30 pages per minute; run a high-level page-description language (PDL); and prove cost-effective for the end user. That set of performance specifications calls for a 32-bit processor—such as Intel’s 80960KB—intended expressly for embedded applications, with good speed and the ability to address large amounts of inexpensive memory (DRAM), on-board floating-point calculations, a large register set, and an on-board interrupt controller.

Speed, of course, translates to fast data movement. In the case of the 80960KB, the burst bus allows speedy interpretation of the PDL, construction of print fonts on-the-fly, and production of printed pages at the maximum rate. The 80960KB yields a high-density, high-performance BITBLT routine at 59 Mbits/sec, using only 80 bytes of memory, and fitting completely in the instruction cache. A single 32- × 40-bit map character BITBLT executes in 472 clocks, or about 6.4% of the 80960KB’s processing time for one page. There is 4 Gbytes of memory space for print fonts and designing in display buffers, I/O, and font caches.

Using slower DRAMs holds down the overall system cost. A large register set, along with register caching, also results in higher performance in moving and rasterizing data. The floating-point processor, with its extensive instruction set, takes care of operations such as font sizing and rotation. Finally, the on-chip interrupt controller provides more efficient communications with both the host computer and the print engine itself.

Page Language Considerations

Placing the PDL and the font descriptions on the controller board makes downloading unnecessary. The board and host can communicate via an RS-232 serial port; the interface between the board and the print engine need not be complicated. Page buffering will help to meet the page-rate goal. While the print engine works on one page, the controller can process the next. To do that, the print engine interface must be buffered, and the software driving the interface must be able to access the page bit-map buffer to feed raster data to the interface. This calls for extensive memory.

To begin with, at least 4 Mbytes of main memory (100-nsec DRAMs) is necessary. To achieve 300 dots/inch, each page takes 1 Mbyte. Therefore, the page bit-map buffer holds 2 Mbytes; font caching and a scratchpad occupy the other two. To hold a language like PostScript on board requires 512 Kbytes of EPROM, which can be implemented with four of Intel’s 27010 1-Mbit (128K × 8) chips.

To handle the approximately 8.5 million data bits on each page, as well as interpret the PDL commands, the Intel 80960KB has a clock of 20 MHz (7.5 MIPS) and burst-bus data rates of 53.3 Mbytes/sec. The overall controller (Figure 1) contains only six logical blocks: CPU, clock generator and reset, bus control, memory, I/O ports, and the print engine. The CPU and control logic consist of the 80960KB, pull-ups for the open-drain signals, address latches, and data transceivers. The 80960KB directly
Figure 6: The print-engine interface for the 80960KB-based design consists of several logical blocks: MDLY (margin delay), PECTL (print-engine control), PRNCR (print-word control register), FIFO (64 words deep), and a shift register. When a page is to be printed, the part executes 16-quad store instructions, filling the FIFO.

words deep), and a shift register. The print controller is interrupt-driven. When a page is ready to be printed, the 80960KB executes 16-quad store instructions, filling the FIFO. When the FIFO goes below eight words, the interrupt handler must fill the FIFO again. Once printing is enabled and the page and line syncs have been asserted, a 16-bit short word loads into the shift register and printing begins.

The interface to the print engine carries six signals: VD (video data), PSYN (page sync), LSYN (line sync), PERDY (print engine ready), PCRDY (print controller ready), and VCLK (video clock). The print-engine controller supplies VD and PCRDY, and the print engine itself, the PSYN, LSYN, PERDY, and VCLK signals. Note that signal definition varies with print engines. Some engines, the Canon SX for one, require an external controller to drive the vertical sync. Others, like the Ricoh engine, supply not only the vertical and horizontal syncs but the video clock. If a print engine does not supply these signals, they can be generated with relative ease using PALs and counters.

A video clock of 4.21 MHz achieves 30-page/min performance. This, however, is much slower than the 80960KB system clock, so that the shift register can be loaded during a shift-out cycle without slowing the printing. An interrupt occurs every 7600 cycles of the 80960KB (200 printed bytes). Since it takes about 176 clocks for the interrupt routine (85 clocks for interrupt latency, 67 clocks for 14 quad stores and one store, and a buffer of 24 clocks in case of cache miss), less than 3% of the 80960KB's time is spent feeding the video port.

A print engine requires an external controller for stepper-motor commands, vertical and horizontal synchronization, and data. The engine receives the rasterized video data from the controller and places a line of charge on the printing cylinder in the raster image of that line from the document. The controller then signals the stepper motor to rotate the cylinder one line, and the next line is charged. This continues for the rest of the page or section of page that is handled in one rotation. The cylinder passes over the toner; the locations that are charged attract toner to the cylinder, which then passes over the highly charged (about 2000V) paper. The charge on the paper attracts the toner from the cylinder. The paper then passes through heated rollers, which fuse the toner.

Describing a Page in Software
Adobe Systems' popular Postscript is a stack-oriented language that uses Postfix notation. Since its introduction, there has been a plethora of similar page description languages introduced to the marketplace. And about 40 companies have announced compatibility with Postscript.

Some page description languages take advantage of the 4-MWhetstone floating-point capabilities of the 80960KB, which significantly adds to performance. About 20% to 30% of processing time is spent executing 64-bit precision floating-point instructions for rotation and translation algorithms and the like. However, for lower-cost designs, the 80960KA, a pin-compatible version without floating point, is perfectly adequate.

ESD:
and the other counter from the word address bits, LAD(2:3), which must be incremented for each subsystem word access (except DRAMs with internal counters). These 2-bit counters can fit easily into one 16R6D PAL.

The address word up-counter generates the word-addressing bits, A(2:3), for the address bus. Loaded with LAD(2:3), this counter can start on any word address boundary, counting up until the burst access is finished or until it reaches a 16-byte boundary. Because bursts cannot cross 16-byte boundaries, the up-counter does not wrap around. If a burst is a two-word access that starts at word address 1, then the word address counter initializes at 1, counts to 2, then stops.

If a burst is four words and starts on a word address other than on a 16-byte boundary, the 80960KB will issue an access with the size bits set so that the access cannot cross 16-byte boundaries. Then it will issue another access, with appropriate size bits to finish the original burst request. The system clock and READY signals enable the word address and burst size counters. By issuing the appropriate size bits to maintain the 16-byte boundary condition, the 80960KB intelligent bus interface helps simplify the burst control design.

Where memory subsystems (Figure 4) are concerned, a language like PostScript requires 350 to 450 Kbytes; 512 Kbytes, plus some initialization code, will be sufficient, and four 27010s will do the job. Because these have a 200-nsec access time, it's necessary to use 2-2-2-2 wait-state timing—two being the number of wait states necessary for each data cycle in a four-word burst access.

### Holding Fonts in Flash

By holding font information in flash memory devices (e.g., the 28F256 32K × 8), fonts can be modified or updated very quickly. The nonvolatile flash is similar to EPROM except that to update the memory, the 80960KB simply writes a command to the control register, then starts writing new data. The 8F256's 170-nsec access translates into 3-3-3-3 wait-state timing. The fonts are accessed, then cached in faster memory by the PDL driver. Three signals control this device: FSHSEL#, RD#, and WR#.

Because two font descriptions fit into 128 Kbytes, 32-bit memory subsystems can be configured from only four flash devices. The most common fonts require about 50 Kbytes/typeface. Software can simply add new fonts by writing the new font descriptions to flash.
The 80960KB's bus performs well with 100-nsec DRAMs, and taking advantage of nibble-mode DRAMs is quite easy because the bus accommodates a four-word burst. The first word takes up the 100-nsec access time; however, subsequent words only take 25 nsec. Two wait states on the first access allow the DRAM row and column decoders to be set up. Subsequent accesses require only the cycling of CAS, which increments the internal column address counter and enables the output drivers. The result: accesses with 0-0-0-0 wait states.

If an access occurs during a refresh sequence, the READY generator simply inserts wait states until the refresh is completed. The DRAM control generates the DRAM READY signal, DRAMRDY#, which is then ORed into the 80960KB's READY signal.

Eight-bit peripherals connect easily to the 80960KB (Figure 5). Although they are slow in comparison, wait states can be inserted to allow for the long access times. The 80960KB's byte-load and byte-store instructions make it quite easy to write 8-bit device drivers. A store byte to the 82510 serial port controller will place the 8-bit value on the 80960KB's AD bus during the data cycle. Data is then held on the bus until the READY# signal returns, indicating the end of the access. The 82510 simply requires that the lower 8 bits of the data bus be tied directly to its data pins. The 82510 uses the same read and write signals as the rest of the system. The READY generation logic controls the number of wait states inserted, and the decode logic generates the

Some PDLs take advantage of the 4-MWhtetstone floating-point capabilities of the 80960KB, which adds to performance.

82510 chip select, IOSEL#.

The 80960KB initiates serial communication with the 82510. When the 82510 receives data, an interrupt is generated. The 80960KB reads the data and places it into memory. To work with other kinds of ports (e.g., Centronics), the port should be assigned a location in memory, the chip select and data lines supplied, and the appropriate number of wait states generated.

Controlling a 30-page/min Print Engine

The print-engine interface (Figure 6) for the 80960KB-based design interface box is also straightforward. It consists of several logical blocks: MDLY (margin delay), PECTL (print-engine control), PRNCR (print-engine control register), FIFO (64

Figure 5: Using an 82510 serial controller, eight-bit peripherals readily connect to the 80960KB. The 80960KB's byte-load and byte-store instructions make it quite easy to write 8-bit device drivers.
To update flash, it must be erased, then written, a word at a time. To accomplish this, one command erases the memory, then another indicates that a data write will follow; data is then written. Since the flash is accessed in 4-byte-wide words, data can be updated each write cycle. Flash memory adds the simplicity of downloading new fonts to the convenience of nonvolatility at power-down.

Because the controller board handles standard 300-dot/inch $8\frac{1}{2}" \times 11"$ pages, a one-page bit-map requires no more than 1 Mbyte of memory. With memory for two buffers, the 80960KB can continue processing the next page while the print engine works on the first. Font caching also places demands on memory, requiring about 128 Kbytes. This amount allows the PDL interpreter to cache previously constructed fonts in the faster memory. Font caching and bit-map buffering help improve performance by reducing the delays caused by slow memory and the print-engine interface.

As far as the DRAMs are concerned, a standard control scheme is appropriate. A DRAM controller takes care of three functions: RAS/CAS cycling, address multiplexing, and refresh timing. A 22VIOB PAL can perform the RAS/CAS cycling for single and multiple reads and writes. It also supplies the control signals for the address multiplexer. The refresh logic, a counter implemented in a PAL, signals the 22VIOB that a refresh cycle is needed; at the end of the current access the control logic starts the refresh sequence.

Main memory consists of 4 Mbytes of 1-Mbit × 1-bit DRAMs. Since no banking is required, only one set of control signals needs to be generated. To provide 1 Mword of memory, 32 parts are needed. Bytes or short words are accessed when the four CAS signals are asserted by the byte enables, LBE# (0:3). One CAS signal can be assigned to each byte in the 32-bit word.
generates control signals for the address latches and data transceivers. With the exception of an inverter for ALE, no glue logic is needed for these signals.

The bus control block implements the chip-select logic and generates the READY signal as well as controlling the 80960KB's burst bus. Bus logic increments the address during a burst access and keeps track of the number of words in the current burst access. It can be implemented in PALs.

Enough EPROM to hold a PDL is contained in the memory block. The block also includes DRAM for page buffering and font caching, as well as flash EPROM to hold the fonts; DRAM control is another part of this logic. The I/O port logic block consists of an RS-232 serial port and an 82510 serial controller. The port drives one DB9 serial connector.

The print engine interface is generic, specified at 30 pages/min; it is relatively straightforward to change the design to fit a specific print engine. An 80960KB interrupt controls the print-engine interface. The logic shown for the interrupt control is a simple 32-bit, 64-word-deep I/O buffer, and could just as well be a 16- or 8-bit printer port with a simple change of 80960KB code. A single-chip printer controller, such as the WD65CIO, can also be designed into an 80960KB or 80960KA system as memory-mapped I/O.

Currently, the 80960KB comes in a PGA package with 132 pins. Thirty-two address/data lines and six signals directly control the bus, resulting in a simple interface. Four byte enables indicate valid bytes in the 32-bit data word; interrupts are signaled via four pins on the 80960KB. Several miscellaneous pins (Figure 2) handle reset, bus master arbitration, bad system access, and clocking. All of the control signals for the address/data bus are open-drain signals and require external pull-ups.

Burst buses often complicate a system design. However, by reducing burst accesses to a maximum of four words per access and supplying the control signals necessary for burst control, the 80960KB holds down the amount of external control.

**The Burst Bus**

Two tasks are necessary to control a burst bus, as shown by the flowchart in Figure 3. One keeps track of the words remaining in the burst access; the second increments the address for each word accessed in the burst. All that’s needed for each of these tasks is a single 2-bit counter. Load one counter from the size bits, LAD(0:1), which indicate how many words are to be accessed.
A Programmer's View of the 80960 Architecture

S. McGeady

Intel Corporation, Hillsboro, OR

ABSTRACT

Intel Corporation's new 80960 processor integrates many architectural features normally found in RISC processors with others found in more traditional architectures. The result is a processor providing high performance while presenting few difficulties for either applications or compiler writers. This paper discusses the programming model of the 960, including aspects of the instruction set and the register architecture. Techniques for effective use of the 960 from both assembly language and high-level languages are discussed, including the subroutine calling sequence designed for the architecture.

1. Introduction

Software engineers interested in the programming model of a processor include application developers, who are primarily interested in the high-level language (HLL) programming model; operating system or kernel developers, who must concern themselves with both the assembly-language programming model and the fault, interrupt, and system-control aspects of the processor; and compiler-writers, who concern themselves with code generation, optimization, and runtime system issues. Compiler and OS developers attempt to insulate the application developer from as many of the details of the architecture as possible. This paper will be of principle interest to those writing assembly-language programs and compiler code-generations or run-time systems for the 960, though a knowledge of the underlying architecture will also be useful for those programming only in a high-level language.

2. The 960 Architecture

This section provides an overview of the 960 architecture. More detail may be found in [Myers88] and a reference manual is available [PRM90].

2.1. Flat Address Space. An engineer developing code in Pascal, C, Ada, or most other Algol-like HLLs will see an extremely simple and straightforward programming model, much like other 32-bit architectures of long standing. The 960 provides a large (4 gigabyte) flat physical address space, with no segments or other limitations on memory addressing. All addresses used by the architecture are 32 bits wide. In implementations that include memory management hardware (currently the 80960MC) standard virtual-memory paging support is supplied, and the virtual address space for each process is also a full 32 bits wide.

The 960 stack may begin at any address in memory, and grows toward higher addresses.

2.2. Fundamental Data Types. Accesses to memory on the 960 can be 8, 16, 32, 64, 96, or 128 bits wide, representing the byte, short, word, longword, tripleword, and quadword types, respectively.

The byte, short, and word data types come in integer (signed) and ordinal varieties. The ld (load) and st (store) operations for bytes and shorts come in each variety, where integer loads sign-extend the most significant bit of the source memory location and ordinal loads do not. Word and wider loads and stores merely copy the sign bit in the normal way, since no sign-extension is required.

Byte ordering within words is little-endian, meaning that the least significant bytes of a word are stored at the lowest-numbered address. This is like the DEC VAX® and Intel 386® processors, but unlike the IBM 360 and Motorola 68000 processors. Future implementations will allow either little-endian or big-endian external memory references.

All current and planned future implementations on the 960 support non-aligned memory accesses, though memory access is fastest when accesses are aligned to natural boundaries, i.e. words to 32-bit boundaries, doubles to 64-bit boundaries, and triples and quads to 128-bit boundaries.

2.3. Register Set. The 960 has a thirty-two general registers and four floating-point registers available to the compiler writer (Fig. 1). The general registers are each 32 bits wide. Twenty-eight of these registers have no predefined function, allowing the compiler great freedom in allocating user procedure-local variables and temporaries into these registers. The remaining four of the 960's 32 general registers are used by the call and ret instructions for stack-pointer, frame-pointer, previous-frame pointer, and return-instruction pointer.

The 960 general registers are divided into two sections: global registers, g0..g15, and local registers, r0..r15. The global registers act like processor registers on any machine, and are affected by instructions only when explicitly used as operands. The local registers are accessible to instructions in exactly the same way as the global registers, but the call instruction provides the called procedure a new set of these registers that, unlike Berkeley RISC [FatSeq81], do not overlap with the previous set. The ret instruction recovers the previous register set for the calling procedure.

The 960 implements a special cache (64 registers in the KA, KB, and MC implementations, up to 192 in a future implementation) for registers containing procedure-local variables, allowing fast procedure call and return. A call instruction causes a new set of 16 local registers to be provided for the called procedure, while the previous procedure's registers are retained in the register cache. Only when the cache is full are registers...
spilled to memory, to locations previously allocated on the stack. This reduces stack accesses due to register spilling during procedure calls to a minimum. Our research (as well as [Ditz82]) shows that most HLL programs tend to execute with far fewer accesses to external memory. This dramatically improves processor performance, especially with moderate-speed memory systems typically found in embedded systems.

The 960 instruction set also allows access to an additional 32 special function registers. In future implementations, these registers will provide access to on-chip peripherals and other special execution units.

2.4. RISC Core Instruction Set. Other than the load (ld), store (st), and a few special-purpose instructions, all instructions in the 960 operate on the general register set. The core instructions of the 960 are:

<table>
<thead>
<tr>
<th>Arithmetic and Logical</th>
<th>Control</th>
<th>Data Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>cmp</td>
<td>branch</td>
</tr>
<tr>
<td>subtract</td>
<td>test</td>
<td>branch-link</td>
</tr>
<tr>
<td>multiply</td>
<td>shift</td>
<td>call</td>
</tr>
<tr>
<td>divide</td>
<td>rotate</td>
<td>return</td>
</tr>
<tr>
<td>modulo</td>
<td>boolean-op</td>
<td></td>
</tr>
</tbody>
</table>

960 instructions are formed from four basic formats (Fig. 2): REG (register) instructions, that form all basic computational instructions, CTRL (control) instructions, including branches and calls, MEM (memory) instructions, the load and store instructions, and COBR (compare-and-branch instructions), a high-density instruction mentioned below.

Fig 2. 960 Instruction Formats.

REG instructions typically take three operands: two source registers and a destination register. Either of the source operands may alternatively be a literal in the range 0..31.

Arithmetic instructions come in ordinal (unsigned) and integer (signed) varieties. These differ in treatment of the most significant bit of the operands, and in the generation of integer overflow faults. Languages such as C that do not define program behaviour on integer overflow typically disable integer overflow fault detection. Languages such as Ada that require integer overflow detection may enable it and do not require additional instructions to check for overflow.

2.5. Integer Arithmetic Controls. The 960 allows detection of overflow during integer arithmetic operations. The integer versions of the arithmetic instructions (addi, subal, muli, divi, etc) may trigger this fault, while the ordinal instructions (add, sub, mul, div, etc) never trigger the fault. The integer overflow trap may be prevented by setting the integer overflow mask in the Arithmetic Controls register. An Integer Zero-Divide Fault is also provided.

2.6. Condition Codes. Most 960 instruction do not set or use the condition codes. In general, only the cmp instructions (and the extended compare instructions discussed below) set the condition codes. The condition codes are contained in the arithmetic controls registers (accessible via the special modac instruction), and are encoded into three bits. Thus, eight masks provide the standard conditions:

<table>
<thead>
<tr>
<th>Branch, Test, and Fault Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>000</td>
</tr>
<tr>
<td>001</td>
</tr>
<tr>
<td>010</td>
</tr>
<tr>
<td>011</td>
</tr>
</tbody>
</table>

The conditional fault instructions also rely on these bits, and the test instructions set their operand register to 1 if the requested condition is true, and 0 if it is false.

The 960 differs from many processors in that the floating-point unit uses the same condition codes (and conditional branch instructions) as the integer unit. The floating-point compare instructions (cmpf, cmpfl) set the condition bits in the manner
described above, except that the never condition indicates that the operands of the compare are unordered, i.e., either operand is an invalid floating-point number such as a NaN. The always condition indicates the ordered condition.

2.7. Extended Instructions. In addition to the core instructions, the 960 implements a set of extended instructions to increase code density, exploit fine-grain parallelism in the microarchitecture, and provide needed functions for embedded applications.

<table>
<thead>
<tr>
<th>Extended Instructions</th>
<th>compare-and-branch</th>
<th>compare-and-increment</th>
<th>compare-and-decrement</th>
<th>conditional-compare</th>
<th>extract bits</th>
<th>modify bits</th>
<th>atomic add</th>
<th>atomic modify</th>
<th>scanbit</th>
<th>spanbit</th>
<th>synchronous move</th>
<th>synchronous load</th>
</tr>
</thead>
</table>

The compare-and-branch and compare-and-increment or decrement instructions exist to improve instruction density by combining typically adjacent instructions when the delay slot between them cannot be filled. This brings the average code density of 960 programs to within 15-25% of that of a VAX, compared to 40% or worse for other RISC processors [Ditz87]. In addition, the conditional-compare instructions are used by the 960 Ada compiler for range checks, e.g:

```
cmpi r0,14  # see if r0 is in the range 14..30
fauline     # fault if it is
```

The `concmpi` instruction acts as a no-op if the result of the previous `cmp` was "less than". This allows simple range checks without conditional branch instructions (and the concomitant pipeline breaks). The atomic and synchronous instructions are important additions for multiprocessor systems.

2.8. Addressing Modes. The 960's load and store instructions provide both the simple, high-performance addressing modes (Fig. 3) normally found in RISC processors, and more complex addressing modes for improved code density and to better exploit fine-grained parallelism in the microarchitecture.

```
<table>
<thead>
<tr>
<th>HLL Code</th>
<th>Assembler Code</th>
<th>Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = global;</td>
<td>Id _global@g0</td>
<td>12 or 32-bit address</td>
</tr>
<tr>
<td>x = &quot;p&quot;;</td>
<td>Id (r6),g0</td>
<td>register-indirect</td>
</tr>
<tr>
<td>x = local;</td>
<td>Id 80(fp),g0</td>
<td>register-indirect + offset</td>
</tr>
<tr>
<td>x = s-&gt;mos;</td>
<td>Id 12(r8),g0</td>
<td>register-indirect + offset</td>
</tr>
<tr>
<td>x = as[i]-&gt;mos;</td>
<td>Id [r6][*4],g0</td>
<td>indexed indirect</td>
</tr>
<tr>
<td>x = as[i]-&gt;mos;</td>
<td>Id [r6][*4][*16],g0</td>
<td>indexed indirect + offset</td>
</tr>
</tbody>
</table>
```

Fig. 3. 950 Memory Addressing Modes

Sophisticated addressing modes on the 960 not only improve code density, but also allow the implementation to compute the effective address of the instruction in parallel with the execution of subsequent instructions. In addition, in each of the above examples, the instruction could have been ld1 (load long), ld0 (load triple), or ldq (load quad), to burst 2, 3, or 4 four words from memory. The instruction 'ldq 12c[9][*4][*16]' would take 6 to 11 instructions to implement on most other RISC processors.

2.9. IEEE-754 Floating-Point. The 960KB implementation includes an on-chip floating-point unit. The FPU is fully IEEE 754-1985 compatible, and supports 32-bit (real), 64-bit (long real), and 80-bit (extended real) precisions. The on-board FPU supports NaN (Not-a-Number), Infinities, Signed Zero, and Denormalized representations, and appropriate (maskable) faults when operations generate NaNs, Infinities, or Denormalized. The FPU also implements four additional 80-bit wide floating-point registers, though floating-point instructions may also operate on the general registers in groups of 1 (real), 2 (long real), or 3 (extended).

```
Floating-Point Instructions
  add binary log
  subtract natural log
  multiply square root
  divide sine
  move cosine
  modulo tangent
  remainder arctangent
```

Each of these instructions can operate on real, long real, or extended precision numbers. The `classify` instruction determines whether a value is a valid floating-point number, or a NaN, Infinity, and/or denormalized, and the sign of the number. The `copySign` operation can be used to provide an absolute value. A full set of conversion instructions are provided that convert between integer and floating-point formats, either using the rounding-mode in effect or truncating.

The 960 FPU can be programmed to fault when it detects denormalized operands, so full IEEE-754 denormalized-number support can be implemented. If normalization mode is on, denormalized numbers are normalized and the operation proceeds without a fault.

2.10. Floating-point Arithmetic Controls. If the HLL or its runtime library supports them, the 960 FPU can provide the following (maskable) flags: floating underflow, overflow, zero-divide, and inexact. The FPU may be set to round up, down, to zero, or to nearest, and may be set in normalizing mode, where denormalized numbers are valid, or non-normalizing mode, where denormalized operands cause a reserved-encoding fault.

The runtime system for C programs typically disables integer overflow, floating underflow, overflow, and inexact faults, and sets the FPU to round-to-nearest and into normalizing mode. The runtime system for Ada programs typically enables all faults and translates them into NUMERIC_ERROR.

3. Subroutine Calling Sequence

No specific calling sequence is mandated or enforced by the 960 architecture. While the call and ret instructions perform pre-defined operations on the stack and frame-pointers, language designers are free to use the `bal` branch-and-link instruction to implement any desired subroutine linkage or calling sequence. A sophisticated compiler might dispense with a standard calling sequence altogether, tuning each call to the needs of the calling and called procedures.

However, Intel has defined a common calling sequence for its C and Ada compilers for the 960. This allows implementation of less sophisticated compilers, assemblers, and debugging tools. Nevertheless, the calling sequence was designed to place an absolute minimum overhead on simple, commonly-called procedures with few parameters, and only slightly more overhead on rarely-used variable-argument-list procedures and procedures

1 The table assumes the following data declarations.
   int global, int array[10],
   local, int i, int p = array,
   struct char a,b,c,d, short e,f,
   float g, int mos,
   )

2 The load and store tripleword functions are provided for loading and storing extended-precision floating-point values to and from memory.
with large numbers of parameters. Intel's 960 software support tools, as well as those supplied for the 960 by most third parties, expect and support this calling sequence.

<table>
<thead>
<tr>
<th>Reg</th>
<th>Primary Use</th>
<th>Secondary Use</th>
<th>P?</th>
</tr>
</thead>
<tbody>
<tr>
<td>g0</td>
<td>parameter 0</td>
<td>return word 0</td>
<td></td>
</tr>
<tr>
<td>g1</td>
<td>parameter 1</td>
<td>return word 1</td>
<td></td>
</tr>
<tr>
<td>g2</td>
<td></td>
<td>return word 2</td>
<td></td>
</tr>
<tr>
<td>g3</td>
<td></td>
<td>return word 3</td>
<td></td>
</tr>
<tr>
<td>g4</td>
<td></td>
<td>reg</td>
<td></td>
</tr>
<tr>
<td>g5</td>
<td></td>
<td>reg</td>
<td></td>
</tr>
<tr>
<td>g6</td>
<td>parameter 6</td>
<td>reg</td>
<td></td>
</tr>
<tr>
<td>g7</td>
<td>parameter 7</td>
<td>reg</td>
<td></td>
</tr>
<tr>
<td>g8</td>
<td>unassigned</td>
<td>parameter 8</td>
<td>P</td>
</tr>
<tr>
<td>g9</td>
<td>unassigned</td>
<td>parameter 9</td>
<td>P</td>
</tr>
<tr>
<td>g10</td>
<td>unassigned</td>
<td>parameter 10</td>
<td>P</td>
</tr>
<tr>
<td>g11</td>
<td>unassigned</td>
<td>parameter 11</td>
<td>P</td>
</tr>
<tr>
<td>g12</td>
<td>unassigned</td>
<td>reg</td>
<td></td>
</tr>
<tr>
<td>g13</td>
<td>structure return ptr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g14</td>
<td>argument blk ptr</td>
<td>leaf return addr</td>
<td></td>
</tr>
<tr>
<td>fp</td>
<td>frame pointer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- P — preserved across calls
- P — preserved if not used as parameter

Fig. 4. Global Register Usage Conventions

3.1. Parameters in Global Registers. The global registers g0..g14 are used for passing parameters and other values between procedures. As shown in Fig. 4, registers g0 through g7 are used for the first eight words of parameters to a procedure. Values are placed into increasing-numbered registers left-to-right, and are aligned within the register set according to their size, possibly leaving holes. A parameter shorter than one word is placed in a single register, two-word parameters (e.g., double-precision floats) are placed in an even-numbered register and the following register, and three and four-word parameters (e.g., extended-precision floats) are placed in g0, g4, or g8. Thus, instructions may use parameters directly from their registers without extracting and aligning them.

Fig 5 shows a C code fragment, and the calling procedure's interface code.

```c
int a, b[10];
... a = foo(a, 1, 'x', &b[0]); ...
    mov r3, #0      # local "a"
    ldconst 1, #1
    ldi 120, #2    # 'x'
    lda 0x40(p), #3 # base of "b"
    call foo
    mov g0, r3
...```

Fig 5. Standard Subroutine Linkage Example

3.2. Return Values in Global Registers. The calling procedure receives any return value shorter than four words in g0..g3 when the called routine returns, allowing integral values, single, double, or extended-precision floating-point values, or small structures to be returned without writing to memory. The calling procedure must assume that the values in g0..g7 are lost across a procedure call (except for those that contain the function return value, if any), though global registers g8..g11 are preserved across the call (the called routine must not modify them, or must save and restore them if they are to be used). Register g12 is always preserved across calls. A globally-optimizing compiler for the 960 could use these registers to hold global constants and pointers to global data structures.

If more than four words of function return value is required, (as in a C function returning a structure) the calling routine must supply a pointer to an area (presumably on the stack) in which the return value is written. If a structure return is needed, a pointer is supplied in register g13, otherwise that register may be used as a temporary.

This linkage convention allows very fast calls with little or no memory traffic to both leaf and non-leaf procedures. A typical non-leaf procedure prologue is:

```c
    _foo:  lda 96(sp),sp # adjust stack
    movq g0,r4 # save parameters
    ret
```

The first instruction (lda — load effective address) adjusts the stack pointer to make room for local (non-register) variables such as arrays and structures. The second instruction copies the incoming four parameters from global registers g0..g3 to local registers r4..r7. Here they will be preserved by the register cache across calls within the procedure. Any procedure can return without adjusting the stack or incurring other overhead by using the ret instruction.

3.3. Support for Argument Blocks. In our examination of many C and Ada programs, we discovered that over 98% of all procedures were called with 6 or fewer parameters [Wei84] ([Pat85] also reports this). However, if more than eight words of parameters are required, four additional words may be placed in g8..g11, and their values, like g4..g7 are indeterminate upon return of the called procedure. If more than 12 words of parameters are required, register g14 is used to point to an argument block on the calling procedure's stack. Registers g0..g11 contain the first twelve words of parameters, and the argument block contains any remaining parameters, following an empty area of twelve words into which the called procedure may copy the parameters passed in the registers. If no argument block is allocated for a procedure, g14 must be set to zero. In practice, procedures with more than 12 words of parameters are so rare that g14 is set in a program's initialization code and seldom changed. Existing compilers typically use the register as a constant zero.

3.4. Variable Length Argument Lists. The C programming language allows procedures to be called with a variable number of arguments. In versions of C before the ANSI X3J11 standardization effort, the calling procedure typically did not know whether a called procedure was a variable-argument-list (varargs) procedure. The 960 calling sequence supports this model, allowing properly-written C programs to be ported without change. The calling procedure follows the rules outlined above, placing parameters in registers until they are exhausted, and then allocating an argument block. The called procedure, however, does not know how many arguments were passed to it and of what type these arguments might be. We rejected the notion of providing an argument count to every procedure, as that would involve undue overhead, and would not solve the type problem.

Register g14 informs the called procedure whether the caller allocated an argument block. If it did, the varargs procedure can simply copy g0..g11 to the stack with three stq (store quad-word) instructions, leaving user code to increment through them. If g14 is set to zero, the caller did not allocate an argument block, and the varargs routine allocates one for itself and copies the parameters into it in the same way. The compiler generates special code in the prologue to varargs routines to do this:
The overhead imposed on varargs routines is minor, and linkage to the preponderance of procedures consists only of a few mov instructions and a call.

3.5. Branch-and-Link Optimizations. Procedures that do not require a stack frame or a set of local registers may be optimized to avoid the allocation of the frame or use of the register cache. Such procedures typically call no other procedures and are called leaf procedures, since they reside at the "leaves" of the call-tree. Entering leaf procedures without creating a new frame makes better use of the 950 register cache and can improve performance in call-intensive programs.

The 960 architecture provides the bal instruction, which branches to the operand address, leaving the address of the subsequent instruction (the return instruction pointer) in a named register. Such a subroutine would return by branching to the address contained in this register.

The 950 compilers generate the calllj (call/jump) pseudo-op in place of the standard call instruction, which allows the linker to determine if separately-compiled modules contain leaf-procedures and promote the call instruction at the call-site to a bal instruction. Fig. 6 shows the entry to a leaf procedure that can also be called in the standard way.

```
leafproc
  __foo: lda ret,g14 # call entry
  __foo: mov g14,g13    # bal entry
  mov 0,g14
  /* remainder of procedure */
  bx (g13)
  ret

Fig. 6 Leaf Procedure Definition
```

3.6. Linkage Conventions for other Languages. The 960 calling convention can be used for C, Ada, Pascal, Fortran, and most other HLLs. Languages with more complex scoping rules than C are sometimes required to pass a static link as an invisi-
ible first (or last) parameter to procedures in enclosed scopes that reference variables in an enclosing scope. The 960 Ada compiler recognizes these cases and passes the static link only to those procedures that require it.

Fortran compilers that implement copy-in/copy-out parameter passing (as opposed to the more common pass-by-reference model) have no problem with SUBROUTINE calls, but FUNCTION calls will require either reference parameters or use of the structure return facility. In Ada, functions do not have in/out parameters, so this is not an issue. Handling unconstrained results in Ada is not contemplated by this linkage convention, but is managed by the 960 Ada compiler in a way that does not violate the convention.

Languages that need not use a standard control stack, or wish to implement a dramatically different calling convention may use branch-and-link exclusively, with a vestigial runtime stack for interrupt and fault handling.

3.7. Trace Controls and Debugging

Many processors are implemented for workstations or end-user computer systems with native operating systems and programming environments. The 960 adds to its architecture a standard set of debugging features that allow simple debugging without a native operating system.

3.8. Trace Controls The 960 implements a series of trace controls (Fig. 7) that allow the user to implement a full runtime debugger as part of a simple monitor implementing the trace fault handler.

```
Fig. 7. 960 Debug Trace Controls
```

The call and return traces allow monitoring of procedure entry and exit, while branch tracing may be used to monitor branch-and-link procedure entries and other branches. The pre-return trace is useful for capturing control immediately before the return from a procedure in order to examine its stack frame. The instruction trace mode faults on the execution of every instruction, allowing single-stepping. Monitors provided by Intel support each of these modes, offering single-stepping, dynamic instruction trace with disassembly. Current implementations of the 960 also include two instruction address breakpoint registers that allow the setting of breakpoints in ROM.

3.9. System Programming

Implementors of operating systems must also concern themselves with the behaviour of interrupts and faults.

3.10. Interrupts. The 960 incorporates a 32-priority, 248-vector interrupt controller on-chip, eliminating the need for off-chip circuitry to handle interrupts. The interrupt table, the location of which is defined at power-up or reinitialization, contains the addresses of the handlers for various interrupts. Also, the first 38 bytes of the interrupt table record the status of all pending interrupts, and all priorities that have pending interrupts.

When an interrupt is received, if it is of a higher priority than other executing or pending interrupts (if any), the processor switches to an independent interrupt stack, saves the arithmetic controls register (containing the condition codes), the process controls register (containing the previously current priority), and the interrupt procedure is called as though from a call instruction, using the handler address in the interrupt table. Since the call instruction automatically allocates a new set of local registers from the register cache, the interrupted procedure’s local variables need not be explicitly saved. Other than the need to save the global registers if they are used, the interrupt service routine is like any other routine, and return from interrupt is effected with the standard ret instruction. The state of the processor, including the previously active priority, is restored when the interrupt returns.

Operating system routines may post software-generated interrupts by using the atomic modify (atmod) instruction to
change values in the pending-interrupts field of the interrupt
table.

3.11. Faults. When the processor detects an exceptional condi-
tion (including "planned" exceptional conditions like trace/debug
faults), a fault is raised. Faults are categorized into trace faults,
invalid operation faults, arithmetic faults, floating-point faults, bad
(memory) access faults, and several processor consistency
faults. Most faults have one or more sub-types that are indicated
when the fault is signaled.

A system-wide fault table contains addresses of fault
handlers for each type of fault. As with interrupts, a fault handler
is entered as though it had been called by the normal call
instruction. Unlike interrupts, however, fault handlers execute on
the user stack, rather than on a separate interrupt stack, allowing
the fault handler simple access to process state information
there. When a fault occurs, a fault record is saved in the fault
handler's stack frame, recording the type and subtype of the
fault, the address of the faulting instruction, the saved arithmetic
and process controls, and sufficient data to restart the instruction
(a resumption record).

For faults that are fatal errors, a fault handler need merely
modify the return instruction pointer in the previous stack frame
(that of the faulting procedure) and return. If it is desired that the
operands of the faulting instruction be modified and the instruc-
tion re-executed, the handler must examine the faulting instruc-
tion, determine the precise cause of the fault, and modify the
operands accordingly.

3.12. Precise, Imprecise, and Parallel Faults. Because the 960
architecture allows instructions to execute in parallel, multiple
faults can occur simultaneously, possibly one or more cycles
after the dispatch of the faulting instruction. If this behaviour
must be avoided, the 960 provides the NIF (No Imprecise Faults)
flag, that prevents parallel execution of instructions that might
generate imprecise faults. However, under normal cir-
cumstances, if multiple faults occur simultaneously, the 960
writes a record for each fault into the stack frame, and calls a
special parallel fault handler (fault type 0). The parallel fault
handler may then dispatch individual fault handlers as appropri-
ate.

For languages such as Ada that require that all potential
faults be signaled at certain places in a procedure (i.e. when an
exception handler is being changed), the synch (synchronize
faults) instruction is provided. This instruction stalls until all
parallel instruction execution units have completed and reported
any faults. The 960 Ada compiler emits this instruction at the end
of an exception frame.

4. Parallel Instruction Optimizations

The 960, like other modern processors, supports pipelined
instruction execution. This allows decode and dispatch opera-
tions for current instructions to be overlapped with execution of
previous instructions. However, the 960 has a fully interlocked
pipeline, ensuring object-code compatibility between current and
future implementations. Unlike many other RISC processors,
there is no need to insert null operations before or after certain
operations such as loads or branches. The 960 also implements
register scoreboarding, ensuring that adjacent instructions that
might be executed in parallel or overlapped do not attempt to use
a single resource at the same time or out of order.

Because of the instruction pipeline careful ordering of
instructions can improve code performance. The result of a ld
instruction may not be available for several cycles after the
instruction is issued, depending on the speed of the memory sys-
tem. By scoreboarding the destination register of the load, the
960 is able to safely continue to execute instructions following the
load, effectively overlapping these instructions with the data fetch
(Fig 8).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld 0x400(r0),g0</td>
<td># these</td>
</tr>
<tr>
<td>mul g4,g5,g6</td>
<td># execute</td>
</tr>
<tr>
<td>addo r0,r1</td>
<td># concurrently</td>
</tr>
<tr>
<td>subo g0,r1</td>
<td># waits for load</td>
</tr>
</tbody>
</table>

Fig. 8. CPU/Bus Parallelism

Because the 960 hardware detects resource conflicts,
software will always operate as expected without the insertion
of null operations, and without software tools to detect and remove
these conflicts.

Future implementations of the 960 will exploit even more
parallelism — a memory operation, an integer operation, and a
branch may be dispatched simultaneously and executed in paral-
el. Careful balancing of memory operations (including lda
instructions, that can perform a limited set of arithmetic opera-
tions) with integer operations can enhance the performance of
future implementations, allowing average 2 instruction per clock
execution rates from on-chip instruction cache.

5. Conclusion

The 960 processor was designed with more than one
implementation in mind. Many features of the 960 are present to
support implementations that provide fine-grained parallelism at
the instruction level, allowing aggregate native instruction rates in
excess of twice the processor clock rate. At the same time, the
960 provides an architecture that is easy to learn and to use, and
that does not require sophisticated software tools to exploit. The
960 combines the practical aspects of RISC techniques
developed in recent years with more traditional mainframe tech-
niques such as register scoreboard and parallel instruction
execution. The calling sequence designed for the 960 allows
enough flexibility to make fast calls to simple non-leaf and leaf
procedures, and yet does not cause undue complication in
development tools. Provisions have been made to support
sophisticated optimizing compilers and other tools as that tech-
nology becomes more mature.

6. References

Cache: Register Allocation for Free", Proceedings
Symposium on Architectural Support for Program-
ming Languages and Operating Systems, March
1982.

[Ditz87] D. Ditzel & H. McLellan, "Design Tradeoffs to Support
the C Programming Language in the CRISP
Microprocessor", Proceedings 2nd International
Conference on Architectural Support for Program-
mimg Languages and Operating Systems, October
1987.


Architecture, Wiley Interscience, New York, NY,


[PatSeq81] D. Patterson & C. Sequin, "RISC I: A Reduced
Instruction Set VLSI Computer," Proceedings 8th
International Symposium on Computer Architecture,
May 1981.

[Weic84] R. Weicker, "Dhrystone: A Synthetic Systems Pro-
gramming Benchmark", CACM, v27n10, October
1984.
Designing Microcontroller Systems for Electrically Noisy Environments

TOM WILLIAMSON
MCO APPLICATIONS ENGINEER
Digital circuits are often thought of as being immune to noise problems, but really they’re not. Noises in digital systems produce software upsets: program jumps to apparently random locations in memory. Noise-induced glitches in the signal lines can cause such problems, but the supply voltage is more sensitive to glitches than the signal lines.

Severe noise conditions, those involving electrostatic discharges, or as found in automotive environments, can do permanent damage to the hardware. Electrostatic discharges can blow a crater in the silicon. In the automotive environment, in ordinary operation, the “12V” power line can show + and −400V transients.

This Application Note describes some electrical noises and noise environments. Design considerations, along the lines of PCB layout, power supply distribution and decoupling, and shielding and grounding techniques, that may help minimize noise susceptibility are reviewed. Special attention is given to the automotive and ESD environments.

**Symptoms of Noise Problems**

Noise problems are not usually encountered during the development phase of a microcontroller system. This is because benches rarely simulate the system’s intended environment. Noise problems tend not to show up until the system is installed and operating in its intended environment. Then, after a few minutes or hours of normal operation the system finds itself someplace out in left field. Inputs are ignored and outputs are gibberish. The system may respond to a reset, or it may have to be turned off physically and then back on again, at which point it commences operating as though nothing had happened. There may be an obvious cause, such as an electrostatic discharge from somebody’s finger to a keyboard or the upset occurs every time a copier machine is turned on or off. Or there may be no obvious cause, and nothing the operator can do will make the upset repeat itself. But a few minutes, or a few hours, or a few days later it happens again.

One symptom of electrical noise problems is randomness, both in the occurrence of the problem and in what the system does in its failure. All operational upsets that occur at seemingly random intervals are not necessarily caused by noise in the system. Marginal VCC, inadequate decoupling, rarely encountered software conditions, or timing coincidences can produce upsets that seem to occur randomly. On the other hand, some noise sources can produce upsets downright periodically. Nevertheless, the more difficult it is to characterize an upset as to cause and effect, the more likely it is to be a noise problem.

**Types and Sources of Electrical Noise**

The name given to electrical noises other than those that are inherent in the circuit components (such as thermal noise) is EMI: electromagnetic interference. Motors, power switches, fluorescent lights, electrostatic discharges, etc., are sources of EMI. There is a veritable alphabet soup of EMI types, and these are briefly described below.

**SUPPLY LINE TRANSIENTS**

Anything that switches heavy current loads onto or off of AC or DC power lines will cause large transients in these power lines. Switching an electric typewriter on or off, for example, can put a 1000V spike onto the AC power lines.

The basic mechanism behind supply line transients is shown in Figure 1. The battery represents any power source, AC or DC. The coils represent the line inductance between the power source and the switchable loads R1 and R2. If both loads are drawing current, the line current flowing through the line inductance establishes a magnetic field of some value. Then, when one of the loads is switched off, the field due to that component of the line current collapses, generating transient voltages, \( v = L(\frac{di}{dt}) \), which try to maintain the current at its original level. That’s called an “inductive kick.” Because of contact bounce, transients are generated whether the switch is being opened or closed, but they’re worse when the switch is being opened.

An inductive kick of one type or another is involved in most line transients, including those found in the automotive environment. Other mechanisms for line transients exist, involving noise pickup on the lines. The noise voltages are then conducted to a susceptible circuit right along with the power.

**EMP AND RFI**

Anything that produces arcs or sparks will radiate electromagnetic pulses (EMP) or radio-frequency interference (RFI).

![Figure 1. Supply Line Transients](image-url)
Spark discharges have probably caused more software upsets in digital equipment than any other single noise source. The upsetting mechanism is the EMP produced by the spark. The EMP induces transients in the circuit, which are what actually cause the upset.

Arcs and sparks occur in automotive ignition systems, electric motors, switches, static discharges, etc. Electric motors that have commutator bars produce an arc as the brushes pass from one bar to the next. DC motors and the "universal" (AC/DC) motors that are used to power hand tools are the kinds that have commutator bars. In switches, the same inductive kick that puts transients on the supply lines will cause an opening or closing switch to throw a spark.

**ESD**

Electrostatic discharge (ESD) is the spark that occurs when a person picks up a static charge from walking across a carpet, and then discharges it into a keyboard, or whatever else can be touched. Walking across a carpet in a dry climate, a person can accumulate a static voltage of 35kV. The current pulse from an electrostatic discharge has an extremely fast risetime — typically, 4A/ns. Figure 2 shows ESD waveforms that have been observed by some investigators of ESD phenomena.

It is enlightening to calculate the L(di/dt) voltage required to drive an ESD current pulse through a couple of inches of straight wire. Two inches of straight wire has about 50 nH of inductance. That's not very much, but using 50 nH for L and 4A/ns for di/dt gives an L(di/dt) drop of about 200V. Recent observations by W.M. King suggest even faster risetimes (Figure 2b) and the occurrence of multiple discharges during a single discharge event.

Obviously, ESD-sensitivity needs to be considered in the design of equipment that is going to be subjected to it, such as office equipment.

**GROUND NOISE**

Currents in ground lines are another source of noise. These can be 60 Hz currents from the power lines, or RF hash, or crosstalk from other signals that are sharing this particular wire as a signal return line. Noise in the ground lines is often referred to as a "ground loop" problem. The basic concept of the ground loop is shown in Figure 3. The problem is that true earth-ground is not really at the same potential in all locations. If the two ends of a wire are earth-grounded at different locations, the voltage difference between the two "ground" points can drive significant currents (several amperes) through the wire. Consider the wire to be part of a loop which contains, in addition to the wire, a voltage source that represents the difference in potential between the two ground points, and you have the classical "ground loop." By extension, the term is used to refer to any unwanted (and often unexpected) currents in a ground line.

**"Radiated" and "Conducted" Noise**

Radiated noise is noise that arrives at the victim circuit in the form of electromagnetic radiation, such as EMP and RFI. It causes trouble by inducing extraneous voltages in the circuit. Conducted noise is noise that arrives at the victim circuit already in the form of an extraneous voltage, typically via the AC or DC power lines.

One defends against radiated noise by care in designing layouts and the use of effective shielding techniques. One defends against conducted noise with filters and
suppressors, although layouts and grounding techniques are important here, too.

**Simulating the Environment**

Addressing noise problems after the design of a system has been completed is an expensive proposition. The ill will generated by failures in the field is not cheap either. It’s cheaper in the long run to invest a little time and money in learning about noise and noise simulation equipment, so that controlled tests can be made on the bench as the design is developing.

Simulating the intended noise environment is a two-step process: First you have to recognize what the noise environment is, that is, you have to know what kinds of electrical noises are present, and which of them are going to cause trouble. Don’t ignore this first step, because it’s important. If you invest in an induction coil spark generator just because your application is automotive, you’ll be straining at the gnat and swallowing the camel. Spark plug noise is the least of your worries in that environment.

The second step is to generate the electrical noise in a controlled manner. This is usually more difficult than first imagined; one first imagines the simulation in terms of a waveform generator and a few spare parts, and then finds that a wideband power amplifier with a 200V dynamic range is also required. A good source of information on who supplies what noise-simulating equipment is the 1981 “ITEM” Directory and Design Guide (Reference 6).

**Types of Failures and Failure Mechanisms**

A major problem that EMI can cause in digital systems is intermittent operational malfunction. These software upsets occur when the system is in operation at the time an EMI source is activated, and are usually characterized by a loss of information or a jump in the execution of the program to some random location in memory. The person who has to iron out such problems is tempted to say the program counter went crazy. There is usually no damage to the hardware, and normal operation can resume as soon as the EMI has passed or the source is de-activated. Resuming normal operation usually requires manual or automatic reset, and possibly re-entering of lost information.

Electrostatic discharges from operating personnel can cause not only software upsets, but also permanent (“hard”) damage to the system. For this to happen the system doesn’t even have to be in operation. Sometimes the permanent damage is latent, meaning the initial damage may be marginal and require further aggravation through operating stress and time before permanent failure takes place. Sometimes too the damage is hidden.

One ESD-related failure mechanism that has been identified has to do with the bias voltage on the substrate of the chip. On some CPU chips the substrate is held at −2.5V by a phase-shift oscillator working into a capacitor/diode clamping circuit. This is called a “charge pump” in chip-design circles. If the substrate wanders too far in either direction, program read errors are noted. Some designs have been known to allow electrostatic discharge currents to flow directly into port pins of an 8048. The resulting damage to the oxide causes an increase in leakage current, which loads down the charge pump, reducing the substrate voltage to a marginal or unacceptable level. The system is then unreliable or completely inoperative until the CPU chip is replaced. But if the CPU chip was subjected to a discharge spark once, it will eventually happen again.

Chips that have a grounded substrate, such as the 8748, can sometimes sustain some oxide damage without actually becoming inoperative. In this case the damage is present, and the increased leakage current is noted; however, since the substrate voltage retains its design value, the damage is largely hidden.

![Figure 3. What a Ground Loop Is](210313-4)
It must therefore be recognized that connecting port pins unprotected to a keyboard or to anything else that is subject to electrostatic discharges, makes an extremely dangerous configuration. It doesn’t make any difference what CPU chip is being used, or who makes it. If it connects unprotected to a keyboard, it will eventually be destroyed. Designing for an ESD-environment will be discussed further on.

We might note here that MOS chips are not the only components that are susceptible to permanent ESD damage. Bipolar and linear chips can also be damaged in this way. PN junctions are subject to a hard failure mechanism called thermal secondary breakdown, in which a current spike, such as from an electrostatic discharge, causes microscopically localized spots in the junction to approach melt temperatures. Low power TTL chips are subject to this type of damage, as are op-amps. Op-amps, in addition, often carry on-chip MOS capacitors which are directly across an external pin combination, and these are susceptible to dielectric breakdown.

We return now to the subject of software upsets. Noise transients can upset the chip through any pin, even an output pin, because every pin on the chip connects to the substrate through a pn junction. However, the most vulnerable pin is probably the VCC line, since it has direct access to all parts of the chip: every register, gate, flip-flop and buffer.

The menu of possible upset mechanisms is quite lengthy. A transient on the substrate at the wrong time will generally cause a program read error. A false level at a control input can cause an extraneous or misdirected opcode fetch. A disturbance on the supply line can flip a bit in the program counter or instruction register. A short interruption or reversal of polarity on the supply line can actually turn the processor off, but not long enough for the power-up reset capacitor to discharge. Thus when the transient ends, the chip starts up again without a reset.

A common failure mode is for the processor to lock itself into a tight loop. Here it may be executing the data in a table, or the program counter may have jumped a notch, so that the processor is now executing operands instead of opcodes, or it may be trying to fetch opcodes from a nonexistent external program memory.

It should be emphasized that mechanisms for upsets have to do with the arrival of noise-induced transients at the pins of the chips, rather than with the generation of noise pulses within the chip itself, that is, it’s not the chip that is picking up noise, it’s the circuit.

The Game Plan
Prevention is usually cheaper than suppression, so first we’ll consider some preventive methods that might help to minimize the generation of noise voltages in the circuit. These methods involve grounding, shielding, and wiring techniques that are directed toward the mechanisms by which noise voltages are generated in the circuit. We’ll also discuss methods of decoupling. Then we’ll look at some schemes for making a graceful recovery from upsets that occur in spite of preventive measures. Lastly, we’ll take another look at two special problem areas: electrostatic discharges and the automotive environment.

Current Loops

The first thing most people learn about electricity is that current won’t flow unless it can flow in a closed loop. This simple fact is sometimes temporarily forgotten by the overworked engineer who has spent the past several years mastering the intricacies of the DO loop, the timing loop, the feedback loop, and maybe even the ground loop. The simple current loop probably owes its apparent demise to the invention of the ground symbol. By a stroke of the pen one avoids having to draw the return paths of most of the current loops in the circuit. Then “ground” turns into an infinite current sink, so that any current that flows into it is gone and forgotten. Forgotten it may be, but it’s not gone. It must return to its source, so that its path will by all the laws of nature form a closed loop.

The physical geometry of a given current loop is the key to why it generates EMI, why it’s susceptible to EMI, and how to shield it. Specifically, it’s the area of the loop that matters.

Any flow of current generates a magnetic field whose intensity varies inversely to the distance from the wire that carries the current. Two parallel wires conducting currents +I and −I (as in signal feed and return lines) would generate a nonzero magnetic field near the wires, where the distance from a given point to one wire is noticeably different from the distance to the other wire, but farther away (relative to the wire spacing), where the distances from a given point to either wire are about the same, the fields from both wires tend to cancel out. Thus, maintaining proximity between feed and return paths is an important way to minimize their interference with other signals. The way to maintain their proximity is essentially to minimize their loop area. And, because the mutual inductance from current loop A to current loop B is the same as the mutual inductance from current loop B to current loop A, a circuit that doesn’t radiate interference doesn’t receive it either.

Thus, from the standpoint of reducing both generation of EMI and susceptibility to EMI, the hard rule is to keep loop areas small. To say that loop areas should be minimized is the same as saying the circuit inductance
should be minimized. Inductance is by definition the constant of proportionality between current and the magnetic field it produces: \( \phi = LI \). Holding the feed and return wires close together so as to promote field cancellation can be described either as minimizing the loop area or as minimizing \( L \). It's the same thing.

**Shielding**

There are three basic kinds of shields: shielding against capacitive coupling, shielding against inductive coupling, and RF shielding. Capacitive coupling is electric field coupling, so shielding against it amounts to shielding against electric fields. As will be seen, this is relatively easy. Inductive coupling is magnetic field coupling, so shielding against it is shielding against magnetic fields. This is a little more difficult. Strangely enough, this type of shielding does not in general involve the use of magnetic materials. RF shielding, the classical "metallic barrier" against all sorts of electromagnetic fields, is what most people picture when they think about shielding. Its effectiveness depends partly on the selection of the shielding material, but mostly, as it turns out, on the treatment of its seams and the geometry of its openings.

**SHIELDING AGAINST CAPACITIVE COUPLING**

Capacitive coupling involves the passage of interfering signals through mutual or stray capacitances that aren't shown on the circuit diagram, but which the experienced engineer knows are there. Capacitive coupling to one's body is what would cause an unstable oscillator to change its frequency when the person reaches his hand over the circuit, for example. More importantly, in a digital system it causes crosstalk in multi-wire cables.

The way to block capacitive coupling is to enclose the circuit or conductor you want to protect in a metal shield. That's called an electrostatic or Faraday shield. If coverage is 100%, the shield does not have to be grounded, but it usually is, to ensure that circuit-to-shield capacitances go to signal reference ground rather than act as feedback and crosstalk elements. Besides, from a mechanical point of view, grounding it is almost inevitable.

A grounded Faraday shield can be used to break capacitive coupling between a noisy circuit and a victim circuit, as shown in Figure 4. Figure 4a shows two circuits capacitively coupled through the stray capacitance between them. In Figure 4b the stray capacitance is intercepted by a grounded Faraday shield, so that interference currents are shunted to ground. For example, a grounded plane can be inserted between PCBs (printed circuit boards) to eliminate most of the capacitive coupling between them.

Another application of the Faraday shield is in the electrostatically shielded transformer. Here, a conducting foil is laid between the primary and secondary coils so as to intercept the capacitive coupling between them. If a system is being upset by AC line transients, this type of transformer may provide the fix. To be effective in this application, the shield must be connected to the greenwire ground.

**SHIELDING AGAINST INDUCTIVE COUPLING**

With inductive coupling, the physical mechanism involved is a magnetic flux density \( B \) from some external interference source that links with a current loop in the victim circuit, and generates a voltage in the loop in accordance with Lenz's law: \( v = -NA(dB/dt) \), where in this case \( N = 1 \) and \( A \) is the area of the current loop in the victim circuit.

There are two aspects to defending a circuit against inductive pickup. One aspect is to try to minimize the offensive fields at their source. This is done by minimizing the area of the current loop at the source so as to promote field cancellation, as described in the section on current loops. The other aspect is to minimize the inductive pickup in the victim circuit by minimizing the area of that current loop, since, from Lenz's law, the induced voltage is proportional to this area. So the two aspects really involve the same corrective action: minimize the areas of the current loops. In other words, minimizing the offensiveness of a circuit inherently minimizes its susceptibility.
Shielding against inductive coupling means nothing more nor less than controlling the dimensions of the current loops in the circuit. We must look at four examples of this type of "shielding": the coaxial cable, the twisted pair, the ground plane, and the gridded-ground PCB layout.

The Coaxial Cable—Figure 5 shows a coaxial cable carrying a current I from a signal source to a receiving load. The shield carries the same current as the center conductor. Outside the shield, the magnetic field produced by +I flowing in the center conductor is cancelled by the field produced by −I flowing in the shield. To the extent that the cable is ideal in producing zero external magnetic field, it is immune to inductive pickup from external sources. The cable adds effectively zero area to the loop. This is true only if the shield carries the same current as the center conductor. In the real world, both the signal source and the receiving load are likely to have one end connected to a common signal ground. In that case, should the cable be grounded at one end, both ends, or neither end? The answer is that it should be grounded at both ends. Figure 6a shows the situation when the cable shield is grounded at only one end. In that case the current loop runs down the center conductor of the cable, then back through the common ground connection. The loop area is not well defined. The shield not only does not carry the same current as the center conductor, but it doesn't carry any current at all. There is no field cancellation at all. The shield has no effect whatsoever on either the generation of EMI or susceptibility to EMI. (It is, however, still effective as an electrostatic shield, or at least it would be if the shield coverage were 100%.)

Figure 6b shows the situation when the cable is grounded at both ends. Does the shield carry all of the return current, or only a portion of it on account of the shunting effect of the common ground connection? The answer to that question depends on the frequency content of the signal. In general, the current loop will follow the path of least impedance. At low frequencies, 0 Hz to several kHz, where the inductive reactance is insignificant, the current will follow the path of least resistance. Above a few kHz, where inductive reactance predominates, the current will follow the path of least inductance. The path of least inductance is the path of minimum loop area. Hence, for higher frequencies the shield carries virtually the same current as the center conductor, and is therefore effective against both generation and reception of EMI.

Note that we have now introduced the famous "ground loop" problem, as shown in Figure 7a. Fortunately, a digital system has some built-in immunity to moderate ground loop noise. In a noisy environment, however, one can break the ground loop, and still maintain the shielding effectiveness of the coaxial cable, by inserting an optical coupler, as shown in Figure 7b. What the optical coupler does, basically, is allow us to re-define the signal source as being ungrounded, so that that end of the cable need not be grounded, and still lets the shield carry the same current as the center conductor. Obviously, if the signal source weren't grounded in the first place, the optical coupler wouldn't be needed.

The Twisted Pair—A cheaper way to minimize loop area is to run the feed and return wires right next to each other. This isn't as effective as a coaxial cable in minimizing loop area. An ideal coaxial cable adds zero area to the loop, whereas merely keeping the feed and return wires next to each other is bound to add a finite area.

However, two things work to make this cheaper method almost as good as a coaxial cable. First, real coaxial cables are not ideal. If the shield current isn't evenly distributed around the center conductor at every cross-
section of the cable (it isn’t), then field cancellation external to the shield is incomplete. If field cancellation is incomplete, then the effective area added to the loop by the cable isn’t zero. Second, in the cheaper method the feed and return wires can be twisted together. This not only maintains their proximity, but the noise picked up in one twist tends to cancel out the noise picked up in the next twist down the line. Thus the “twisted pair” turns out to be about as good a shield against inductive coupling as coaxial cable is.

The twisted pair does not, however, provide electrostatic shielding (i.e., shielding against capacitive coupling). Another operational difference between them is that the coaxial cable works better at higher frequencies. This is primarily because the twisted pair adds more capacitive loading to the signal source than the coaxial cable does. The twisted pair is normally considered useful up to only about 1 MHz, as opposed to near a GHz for the coaxial cable.

The Ground Plane—The best way to minimize loop areas when many current loops are involved is to use a ground plane. A ground plane is a conducting surface that is to serve as a return conductor for all the current loops in the circuit. Normally, it would be one or more layers of a multilayer PCB. All ground points in the circuit go not to a grounded trace on the PCB, but directly to the ground plane. This leaves each current loop in the circuit free to complete itself in whatever configuration yields minimum loop area (for frequencies wherein the ground path impedance is primarily inductive).

Thus, if the feed path for a given signal zigzags its way across the PCB, the return path for this signal is free to zigzag right along beneath it on the ground plane, in such a configuration as to minimize the energy stored in the magnetic field produced by this current loop. Minimal magnetic flux means minimal effective loop area and minimal susceptibility to inductive coupling.

The Gridded-Ground PCB Layout—The next best thing to a ground plane is to layout the ground traces on a PCB in the form of a grid structure, as shown in Figure 8. Laying horizontal traces on one side of the board and vertical traces on the other side allows the passage of signal and power traces. Wherever vertical and horizontal ground traces cross, they must be connected by a feed-through.

Have we not created here a network of “ground loops”? Yes, in the literal sense of the word, but loops in the ground layout on a PCB are not to be feared. Such inoffensive little loops have never caused as much noise pickup as their avoidance has. Trying to avoid innocent little loops in the ground layout, PCB designers have forced current loops into geometries that could swallow a whale. That is exactly the wrong thing to do.

The gridded ground structure works almost as well as the ground plane, as far as minimizing loop area is concerned. For a given current loop, the primary return path may have to zig once in a while where its feed path zags, but you still get a mathematically optimal dis-
distribution of currents in the grid structure, such that the current loop produces less magnetic flux than if the return path were restrained to follow any single given ground trace. The key to attaining minimum loop areas for all the current loops together is to let the ground currents distribute themselves around the entire area of the board as freely as possible. They want to minimize their own magnetic field. Just let them.

**RF SHIELDING**

A time-varying electric field generates a time-varying magnetic field, and vice versa. Far from the source of a time-varying EM field, the ratio of the amplitudes of the electric and magnetic fields is always 377Ω. Up close to the source of the fields, however, this ratio can be quite different, and dependent on the nature of the source. Where the ratio is near 377Ω is called the far field, and where the ratio is significantly different from 377Ω is called the near field. The ratio itself is called the wave impedance, E/H.

The near field goes out about 1/6 of a wavelength from the source. At 1 MHz this is about 150 feet, and at 10 MHz it's about 15 feet. That means if an EMI source is in the same room with the victim circuit, it's likely to be a near field problem. The reason this matters is that in the near field an RF interference problem could be almost entirely due to E-field coupling or H-field coupling, and that could influence the choice of an RF shield or whether an RF shield will help at all.

In the near field of a whip antenna, the E/H ratio is higher than 377Ω, which means it’s mainly an E-field generator. A wire-wrap post can be a whip antenna. Interference from a whip antenna would be by electric field coupling, which is basically capacitive coupling. Methods to protect a circuit from capacitive coupling, such as a Faraday shield, would be effective against RF interference from a whip antenna. A gridded-ground structure would be less effective.

In the near field of a loop antenna, the E/H ratio is lower than 377Ω, which means it’s mainly an H-field generator. Any current loop is a loop antenna. Interference from a loop antenna would be by magnetic field coupling, which is basically the same as inductive coupling. Methods to protect a circuit from inductive coupling, such as a gridded-ground structure, would be effective against RF interference from a loop antenna. A Faraday shield would be less effective.

A more difficult case of RF interference, near field or far field, may require a genuine metallic RF shield. The idea behind RF shielding is that time-varying EMI fields induce currents in the shielding material. The induced currents dissipate energy in two ways: I²R losses in the shielding material and radiation losses as they re-radiate their own EM fields. The energy for both of these mechanisms is drawn from the impinging EMI fields. Hence the EMI is weakened as it penetrates the shield.

More formally, the I²R losses are referred to as absorption loss, and the re-radiation is called reflection loss. As it turns out, absorption loss is the primary shielding mechanism for H-fields, and reflection loss is the primary shielding mechanism for E-fields. Reflection loss, being a surface phenomenon, is pretty much independent of the thickness of the shielding material. Both loss mechanisms, however, are dependent on the frequency (ω) of the impinging EMI field, and on the permeability (μ) and conductivity (σ) of the shielding material. These loss mechanisms vary approximately as follows:

\[
\text{reflection loss to an E-field (in dB)} \sim \log \frac{\sigma}{\omega \mu} \\
\text{absorption loss to an H-field (in dB)} \sim t \sqrt{\omega \mu \sigma}
\]

where t is the thickness of the shielding material.

The first expression indicates that E-field shielding is more effective if the shield material is highly conductive, and less effective if the shield is ferromagnetic, and that low-frequency fields are easier to block than high-frequency fields. This is shown in Figure 9.
Copper and aluminum both have the same permeability, but copper is slightly more conductive, and so provides slightly greater reflection loss to an E-field. Steel is less effective for two reasons. First, it has a somewhat elevated permeability due to its iron content, and second, as tends to be the case with magnetic materials, it is less conductive.

On the other hand, according to the expression for absorption loss to an H-field, H-field shielding is more effective at higher frequencies and with shield material that has both high conductivity and high permeability. In practice, however, selecting steel for its high permeability involves some compromise in conductivity. But the increase in permeability more than makes up for the decrease in conductivity, as can be seen in Figure 10. This figure also shows the effect of shield thickness.

A composite of E-field and H-field shielding is shown in Figure 11. However, this type of data is meaningful only in the far field. In the near field the EMI could be 90% H-field, in which case the reflection loss is irrelevant. It would be advisable then to beef up the absorption loss, at the expense of reflection loss, by choosing steel. A better conductor than steel might be less expensive, but quite ineffective.

A different shielding mechanism that can be taken advantage of for low frequency magnetic fields is the ability of a high permeability material such as mumetal to divert the field by presenting a very low reluctance path to the magnetic flux. Above a few kHz, however, the permeability of such materials is the same as steel.

In actual fact the selection of a shielding material turns out to be less important than the presence of seams, joints and holes in the physical structure of the enclosure. The shielding mechanisms are related to the induction of currents in the shield material, but the currents must be allowed to flow freely. If they have to detour around slots and holes, as shown in Figure 12, the shield loses much of its effectiveness.

As can be seen in Figure 12, the severity of the detour has less to do with the area of the hole than it does with the geometry of the hole. Comparing Figure 12c with 12d shows that a long narrow discontinuity such as a seam can cause more RF leakage than a line of holes with larger total area. A person who is responsible for designing or selecting rack or chassis enclosures for an EMI environment needs to be familiar with the techniques that are available for maintaining electrical continuity across seams. Information on these techniques is available in the references.

**Grounds**

There are two kinds of grounds: earth-ground and signal ground. The earth is not an equipotential surface, so earth ground potential varies. That and its other electrical properties are not conducive to its use as a return conductor in a circuit. However, circuits are often connected to earth ground for protection against shock hazards. The other kind of ground, signal ground, is an arbitrarily selected reference node in a circuit—the node with respect to which other node voltages in the circuit are measured.

**SAFETY GROUND**

The standard 3-wire single-phase AC power distribution system is represented in Figure 13. The white wire is earth-grounded at the service entrance. If a load circuit has a metal enclosure or chassis, and if the black wire develops a short to the enclosure, there will be a shock hazard to operating personnel, unless the enclosure itself is earth-grounded. If the enclosure is earth-
grounded, a short results in a blown fuse rather than a “hot” enclosure. The earth-ground connection to the enclosure is called a safety ground. The advantage of the 3-wire power system is that it distributes a safety ground along with the power.

Note that the safety-ground wire carries no current, except in case of a fault, so that at least for low frequencies it’s at earth-ground potential along its entire length. The white wire, on the other hand, may be several volts off ground, due to the IR drop along its length.

SIGNAL GROUND

Signal ground is a single point in a circuit that is designated to be the reference node for the circuit. Commonly, wires that connect to this single point are also referred to as “signal ground.” In some circles “power supply common” or PSC is the preferred terminology for these conductors. In any case, the manner in which these wires connect to the actual reference point is the basis of distinction among three kinds of signal-ground wiring methods: series, parallel, and multipoint. These methods are shown in Figure 14.

The series connection is pretty common because it’s simple and economical. It’s the noisiest of the three, however, due to common ground impedance coupling between the circuits. When several circuits share a ground wire, currents from one circuit, flowing through the finite impedance of the common ground line, cause variations in the ground potential of the other circuits. Given that the currents in a digital system tend to be spiked, and that the common impedance is mainly inductive reactance, the variations could be bad enough to cause bit errors in high current or particularly noisy situations.

The parallel connection eliminates common ground impedance problems, but uses a lot of wire. Other disadvantages are that the impedance of the individual ground lines can be very high, and the ground lines themselves can become sources of EMI.
In the multipoint system, ground impedance is minimized by using a ground plane with the various circuits connected to it by very short ground leads. This type of connection would be used mainly in RF circuits above 10 MHz.

PRACTICAL GROUNDING

A combination of series and parallel ground-wiring methods can be used to trade off economic and the various electrical considerations. The idea is to run series connections for circuits that have similar noise properties, and connect them at a single reference point, as in the parallel method, as shown in Figure 15.

In Figure 15, "noisy signal ground" connects to things like motors and relays. Hardware ground is the safety ground connection to chassis, racks, and cabinets. It's a mistake to use the hardware ground as a return path for signal currents because it's fairly noisy (for example, it's the hardware ground that receives an ESD spark) and tends to have high resistance due to joints and seams.

![Figure 15. Parallel Connection of Series Grounds](image)

Screws and bolts don't always make good electrical connections because of galvanic action, corrosion, and dirt. These kinds of connections may work well at first, and then cause mysterious maladies as the system ages.

Figure 16 illustrates a grounding system for a 9-track digital tape recorder, showing an application of the series/parallel ground-wiring method.

Figure 17 shows a similar separation of grounds at the PCB level. Currents in multiplexed LED displays tend to put a lot of noise on the ground and supply lines because of the constant switching and changing involved in the scanning process. The segment driver ground is relatively quiet, since it doesn't conduct the LED currents. The digit driver ground is noisier, and should be provided with a separate path to the PCB ground terminal, even if the PCB ground layout is gridded. The LED feed and return current paths should be laid out on opposite sides of the board like parallel flat conductors.

Figure 18 shows right and wrong ways to make ground connections in racks. Note that the safety ground connections from panel to rack are made through ground straps, not panel screws. Rack 1 correctly connects signal ground to rack ground only at the single reference point. Rack 2 incorrectly connects signal ground to rack ground at two points, creating a ground loop around points 1, 2, 3, 4, 1.

Breaking the "electronics ground" connection to point 1 eliminates the ground loop, but leaves signal ground in rack 2 sharing a ground impedance with the relatively noisy hardware ground to the reference point; in fact, it may end up using hardware ground as a return path for signal and power supply currents. This will probably cause more problems than the ground loop.

BRAIDED CABLE

Ground impedance problems can be virtually eliminated by using braided cable. The reduction in impedance is due to skin effect: At higher frequencies the current tends to flow along the surface of a conductor rather
Figure 16. Ground System in a 9-Track Digital Recorder

Figure 17. Separate Ground for Multiplexed LED Display
than uniformly through its bulk. While this effect tends to increase the impedance of a given conductor, it also indicates the way to minimize impedance, and that is to manipulate the shape of the cross-section so as to provide more surface area. For its bulk, braided cable is almost pure surface.

**Power Supply Distribution and Decoupling**

The main consideration for power supply distribution lines is, as for signal lines, to minimize the areas of the current loops. But the power supply lines take on an importance that no signal line has when one considers the fact that these lines have access to every PC board in the system. The very extensiveness of the supply current loops makes it difficult to keep loop areas small. And, a noise glitch on a supply line is a glitch delivered to every board in the system.

The power supply provides low-frequency current to the load, but the inductance of the board-to-board and chip-to-chip distribution network makes it difficult for the power supply to maintain VCC specs on the chip while providing the current spikes that a digital system requires. In addition, the power supply current loop is a very large one, which means there will be a lot of noise pick-up. Figure 19a shows a load circuit trying to draw current spikes from a supply voltage through the line impedance. To the VCC waveform shown in that figure should be added the inductive pick-up associated with a large loop area.

Adding a decoupling capacitor solves two problems: The capacitor acts as a nearby source of charge to supply the current spikes through a smaller line impedance, and it defines a much smaller loop area for the higher frequency components of EMI. This is illustrated in Figure 19b, which shows the capacitor supplying the current spike, during which VCC drops from 5V by the amount, indicated in the figure. Between current spikes the capacitor recovers through the line impedance.

One should resist the temptation to add a resistor or an inductor to the decoupler so as to form a genuine RC or LC low-pass filter because that slows down the speed with which the decoupler cap can be refreshed. Good filtering and good decoupling are not necessarily the same thing.

The current loop for the higher frequency currents, then, is defined by the decoupling cap and the load circuit, rather than by the power supply and the load circuit. For the decoupling cap to be able to provide the current spikes required by the load, the inductance of this current loop must be kept small, which is the same as saying the loop area must be kept small. This is also the requirement for minimizing inductive pick-up in the loop.

There are two kinds of decoupling caps: board decouplers and chip decouplers. A board decoupler will normally be a 10 to 100 μF electrolytic capacitor placed near to where the power supply enters the PC board, but its placement is relatively non-critical. The purpose of the board decoupler is to refresh the charge on the chip decouplers. The chip decouplers are what actually provide the current spikes to the chips. A chip decoupler will normally be a 0.1 to 1 μF ceramic capacitor placed near the chip and connected to the chip by traces that minimize the area of the loop formed by the cap and the chip. If a chip decoupler is not properly placed on the board, it will be ineffective as a decoupler.
and will serve only to increase the cost of the board. Good and bad placement of decoupling capacitors are illustrated in Figure 20.

Power distribution traces on the PC board need to be laid out so as to obtain minimal area (minimal inductance) in the loops formed by each chip and its decoupler, and by the chip decouplers and the board decoupler. One way to accomplish this goal is to use a power plane. A power plane is the same as a ground plane, but at VCC potential. More economically, a power grid similar to the ground grid previously discussed (Figure 8) can be used. Actually, if the chip decoupling loops are small, other aspects of the power layout are less critical. In other words, power planes and power gridding aren’t needed, but power traces should be laid in the closest possible proximity to ground traces, preferably so that each power trace is on the direct opposite side of the board from a ground trace.

Special-purpose power supply distribution buses which mount on the PCB are available. The buses use a parallel flat conductor configuration, one conductor being a VCC line and the other a ground line. Used in conjunction with a gridded ground layout, they not only provide a low-inductance distribution system, but can themselves form part of the ground grid, thus facilitating the PCB layout. The buses are available with and without enhanced bus capacitance, under the names Mini/Bus® and Q/PAC® from Rogers Corp. (5750 E. McKellips, Mesa, AZ 85205).

SELECTING THE VALUE OF THE DECOUPLING CAP

The effectiveness of the decoupling cap has a lot to do with the way the power and ground traces connect this capacitor to the chip. In fact, the area formed by this loop is more important than the value of the capacitance. Then, given that the area of this loop is indeed minimal, it can generally be said that the larger the value of the decoupling cap, the more effective it is, if the cap has a mica, ceramic, glass, or polystyrene dielectric.

It’s often said, and not altogether accurately, that the chip decoupler shouldn’t have too large a value. There are two reasons for this statement. One is that some capacitors, because of the nature of their dielectrics, tend to become inductive or lossy at higher frequencies. This is true of electrolytic capacitors, but mica, glass,
ceramic, and polystyrene dielectrics work well to several hundred MHz. The other reason cited for not using too large a capacitance has to do with lead inductance.

The capacitor with its lead inductance forms a series LC circuit. Below the frequency of series resonance, the net impedance of the combination is capacitive. Above that frequency, the net impedance is inductive. Thus a decoupling capacitor is capacitive only below the frequency of series resonance. The frequency is given by

\[ f_0 = \frac{1}{2\pi\sqrt{LC}} \]

where \( C \) is the decoupling capacitance and \( L \) is the lead inductance between the capacitor and the chip. On a PC board this inductance is determined by the layout, and is the same whether the capacitor dropped into the PCB holes is 0.001 \( \mu \)F or 1 \( \mu \)F. Thus, increasing the capacitance lowers the series resonant frequency. In fact, according to the resonant frequency formula, increasing \( C \) by a factor of 100 lowers the resonant frequency by a factor of 10.

Figures quoted on the series resonant frequency of a 0.01 \( \mu \)F capacitor run from 10 to 15 MHz, depending on the lead length. If these numbers were accurate, a 1 \( \mu \)F capacitor in the same position on the board would have a resonant frequency of 1.0 to 1.5 MHz, and as a decoupler would do more harm than good. However, the numbers are based on a presumed inductance of a given length of wire (the lead length). It should be noted that a "length of wire" has no inductance at all, strictly speaking. Only a complete current loop has inductance, and the inductance depends on the geometry of the loop. Figures quoted on the inductance of a length of wire are based on a presumably "very large" loop area, such that the magnetic field produced by the return current has no cancellation effect on the field produced by the current in the given length of wire. Such a loop geometry is not and should not be the case with the decoupling loop.

Figure 21 shows VCC waveforms, measured between pins 40 and 20 (VCC and VSS) of an 8751 CPU, for several conditions of decoupling on a PC board that has a decoupling loop area slightly larger than necessary. These photographs show the effects of increasing the decoupling capacitance and decreasing the area of the decoupling loop. The indications are that a 1 \( \mu \)F capacitor is better than a 0.1 \( \mu \)F capacitor, which in turn is better than nothing, and that the board should have been laid out with more attention paid to the area of the decoupling loop.

Figure 21e was obtained using a special-purpose experimental capacitor designed by Rogers Corp. (Q-Pac Division, Mesa, AZ) for use as a decoupler. It consists of two parallel plates, the length of a 40-pin DIP, separated by a ceramic dielectric. Sandwiched between the CPU chip and the PCB (or between the CPU socket and the PCB), it makes connection to pins 40 and 20, forming a leadless decoupling capacitor. It is obviously a configuration of minimal inductance. Unfortunately, the particular sample tested had only 0.07 \( \mu \)F of capacitance and so was unable to prevent the 1 MHz ripple as effectively as the configuration of Figure 21d. It seems apparent, though, that with more capacitance this part will alleviate a lot of decoupling problems.

THE CASE FOR ON-BOARD VOLTAGE REGULATION

To complicate matters, supply line glitches aren't always picked up in the distribution networks, but can come from the power supply circuit itself. In that case, a well-designed distribution network faithfully delivers the glitch throughout the system. The VCC glitch in Figure 22 was found to be coming from within a bench power supply in response to the EMP produced by an induction coil spark generator that was being used at Intel during a study of noise sensitivity. The VCC glitch is about 400 mV high and some 20 \( \mu \)s in duration. Normal board decoupling techniques were ineffective in removing it, but adding an on-board voltage regulator chip did the job.

Thus, a good case can be made in favor of using a voltage regulator chip on each PCB, instead of using all the voltage regulation at the supply circuit. This eases requirements on the heat-sinking at the supply circuit, and alleviates much of the distribution and board decoupling headaches. However, it also brings in the possibility that different boards would be operating at slightly different VCC levels due to tolerance in the regulator chips; this then leads to slightly different logic levels from board to board. The implications of that may vary from nothing to latch-up, depending on what kinds of chips are on the boards, and how they react to an input "high" that is perhaps 0.4V higher than local VCC.

Recovering Gracefully from a Software Upset

Even when one follows all the best guidelines for designing for a noisy environment, it's always possible for a noise transient to occur which exceeds the circuit's immunity level. In that case, one can strive at least for a graceful recovery.

Graceful recovery schemes involve additional hardware and/or software which is supposed to return the system to a normal operating mode after a software upset has occurred. Two decisions have to be made: How to recognize when an upset has occurred, and what to do about it.
(a) No Decoupling Cap

(b) 0.1 µF Decoupler in Place on the PCB

(c) 0.1 µF Decoupler Stretched Directly from Pin 40 to Pin 20, under the Socket. (The difference between this and 21b is due only to the change in loop geometry. Also shown is the upward slope of a ripple in VCC. The ripple frequency is 1 MHz, the same as ALE.)

(d) 1.0 µF Decoupler Stretched Directly from Pin 40 to Pin 20, under the Socket. (This prevents the 1 MHz ripple, but there's no reduction in higher frequency components. Further increases in capacitance effected no further improvement.)

(e) Special-Purpose Decoupling Cap under Development by Rogers Corp. (Further discussion in text.)

Figure 21. Noise on VCC Line
If the designer knows what kinds and combinations of outputs can legally be generated by the system, he can use gates to recognize and flag the occurrence of an illegal state of affairs. The flag can then trigger a jump to a recovery routine which then may check or re-initialize data, perhaps output an error message, or generate a simple reset.

The most reliable scheme is to use a so-called watchdog circuit. Here the CPU is programmed to generate a periodic signal as long as the system is executing instructions in an expected manner. The periodic signal is then used to hold off a circuit that will trigger a jump to a recovery routine. The periodic signal needs to be 

ac-coupled to the trigger circuit so that a “stuck-at” fault won’t continue to hold off the trigger. Then, if the processor locks up someplace, the periodic signal is lost and the watchdog triggers a reset.

In practice, it may be convenient to drive the watchdog circuit with a signal which is being generated anyway by the system. One needs to be careful, however, that an upset does in fact discontinue that signal. Specifically, for example, one could use one of the digit drive signals going to a multiplexed display. But display scanning is often handled in response to a timer-interrupt, which may continue operating even though the main program is in a failure mode. Even so, with a little extra software, the signal can be used to control the watchdog (see Reference 8 on this).

Simpler schemes can work well for simpler systems. For example, if a CPU isn't doing anything but scanning and decoding a keyboard, there's little to lose and much to gain by simply resetting it periodically with an astable multivibrator. It only takes about 13 µs (at 6 MHz) to reset an 8048 if the clock oscillator is already running.

A zero-cost measure is simply to fill all unused program memory with NOPs and JMPs to a recovery routine. The effectiveness of this method is increased by writing the program in segments that are separated by NOPs and JMPs. It's still possible, of course, to get hung up in a data table or something. But you get a lot of protection, for the cost.

Further discussion of graceful recovery schemes can be found in Reference 13.

**Special Problem Areas**

**ESD**

MOS chips have some built-in protection against a static charge build-up on the pins, as would occur during normal handling, but there's no protection against the kinds of current levels and rise times that occur in a genuine electrostatic spark. These kinds of discharges can blow a crater in the silicon.

It must be recognized that connecting CPU pins unprotected to a keyboard or to anything else that is subject to electrostatic discharges makes an extremely fragile configuration. Buffering them is the very least one can do. But buffering doesn't completely solve the problem, because then the buffer chips will sustain the damage (even TTL); therefore, one might consider mounting the buffer chips in sockets for ease of replacement.

Transient suppressors, such as the TranZorbs® made by General Semiconductor Industries (Tempe, AZ), may in the long run provide the cheapest protection if their “zero inductance” structure is used. The structure and circuit application are shown in Figure 23.

The suppressor element is a pn junction that operates like a Zener diode. Back-to-back units are available for AC operation. The element is more or less an open circuit at normal system voltage (the standoff voltage rating for the device), and conducts like a Zener diode at the clamping voltage.

The lead inductance in the conventional transient suppressor package makes the conventional package essen-
tially useless for protection against ESD pulses, owing to the fast rise of these pulses. The “zero inductance” units are available singly in a 4-pin DIP, and in arrays of four to a 16-pin DIP for PCB level protection. In that application they should be mounted in close proximity to the chips they protect.

In addition, metal enclosures or frames or parts that can receive an ESD spark should be connected by braided cable to the green-wire ground. Because of the ground impedance, ESD current shouldn’t be allowed to flow through any signal ground, even if the chips are protected by transient suppressors. A 35 kV ESD spark can always spare a few hundred volts to drive a fast current pulse down a signal ground line if it can’t find a braided cable to follow. Think how delighted your 8048 will be to find its VSS pin 250V higher than VCC for a few 10s of nanoseconds.

THE AUTOMOTIVE ENVIRONMENT

The automobile presents an extremely hostile environment for electronic systems. There are several parts to it:

1. Temperature extremes from −40°C to +125°C (under the hood) or +85°C (in the passenger compartment)
2. Electromagnetic pulses from the ignition system
3. Supply line transients that will knock your socks off

One needs to take a long, careful look at the temperature extremes. The allowable storage temperature range for most Intel MOS chips is −65°C to +150°C, although some chips have a maximum storage temperature rating of +125°C. In operation (or “under bias,” as the data sheets say) the allowable ambient temperature range depends on the product grade, as follows:

<table>
<thead>
<tr>
<th>Grade</th>
<th>Ambient Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>Commercial</td>
<td>0</td>
</tr>
<tr>
<td>Industrial</td>
<td>−40</td>
</tr>
<tr>
<td>Automotive</td>
<td>−40</td>
</tr>
<tr>
<td>Military</td>
<td>−55</td>
</tr>
</tbody>
</table>

The different product grades are actually the same chip, but tested according to different standards. Thus, a given commercial-grade chip might actually pass military temperature requirements, but not have been tested for it. (Of course, there are other differences in grading requirements having to do with packaging, burn-in, traceability, etc.)

In any case, it’s apparent that commercial-grade chips can’t be used safely in automotive applications, not even in the passenger compartment. Industrial-grade chips can be used in the passenger compartment, and automotive or military chips are required in under-the-hood applications.

Ignition noise, CB radios, and that sort of thing are probably the least of your worries. In a poorly designed system, or in one that has not been adequately tested for the automotive environment, this type of EMI might cause a few software upsets, but not destroy chips.

The major problem, and the one that seems to come as the biggest surprise to most people, is the line transients. Regrettably, the 12V battery is not actually the source of power when the car is running. The charging system is, and it’s not very clean. The only time the battery is the real source of power is when the car is first being started, and in that condition the battery terminals may be delivering about 5V or 6V. As follows is a brief description of the major idiosyncracies of the “12V” automotive power line.
• An abrupt reduction in the alternator load causes a positive voltage transient called “load dump.” In a load dump transient the line voltage rises to 20V or 30V in a few μs, then decays exponentially with a time constant of about 100 μs, as shown in Figure 24. Much higher peak voltages and longer decay times have also been reported. The worst case load dump is caused by disconnecting a low battery from the alternator circuit while the alternator is running. Normally this would happen intermittently when the battery terminal connections are defective.

• When the ignition is turned off, as the field excitation decays, the line voltage can go to between -40V and -100V for 100 μs or more.

• Miscellaneous solenoid switching transients, such as the one shown in Figure 25, can drive the line to + or -200V to 400V for several μs.

• Mutual coupling between unshielded wires in long harnesses can induce 100V and 200V transients in unprotected circuits.

What all this adds up to is that people in the business of building systems for automotive applications need a comprehensive testing program. An SAE guideline which describes the automotive environment is available to designers: SAE J1211, “Recommended Environmental Practices for Electronic Equipment Design,” 1980 SAE Handbook, Part 1, pp. 22.80–22.96.

Some suggestions for protecting circuitry are shown in Figure 26. A transient suppressor is placed in front of the regulator chip to protect it. Since the rise times in these transients are not like those in ESD pulses, lead inductance is less critical and conventional devices can be used. The regulator itself is pretty much of a necessity, since a load dump transient is simply not going to be removed by any conventional LC or RC filter.
Special I/O interfacing is also required, because of the need for high tolerance to voltage transients, input noise, input/output isolation, etc. In addition, switches that are being monitored or driven by these buffers are usually referenced to chassis ground instead of signal ground, and in a car there can be many volts difference between the two. I/O interfacing is discussed in Reference 2.

The EMC Education committee has available a video tape: "Introduction to EMC—A Video Training Tape," by Henry Ott. Don White Consultants offers a series of training courses on many different aspects of electromagnetic compatibility. Most organizations that sponsor EMC courses also offer in-plant presentations.

### Parting Thoughts

The main sources of information for this Application Note were the references by Ott and by White. Reference 5 is probably the finest treatment currently available on the subject. The other references provided specific information as cited in the text.

Courses and seminars on the subject of electromagnetic interference are given regularly throughout the year. Information on these can be obtained from:

IEEE Electromagnetic Compatibility Society  
EMC Education Committee  
345 East 47th Street  
New York, NY 10017

Don White Consultants, Inc.  
International Training Centre  
P.O. Box D  
Gainesville, VA 22065  
Phone: (703) 347-0030
REFERENCES


Oscillators for Microcontrollers

TOM WILLIAMSON
MICROCONTROLLER
TECHNICAL MARKETING
INTRODUCTION

Intel's microcontroller families (MCS®-48, MCS®-51, and iACX-96) contain a circuit that is commonly referred to as the "on-chip oscillator". The on-chip circuitry is not itself an oscillator, of course, but an amplifier that is suitable for use as the amplifier part of a feedback oscillator. The data sheets and Microcontroller Handbook show how the on-chip amplifier and several off-chip components can be used to design a working oscillator. With proper selection of off-chip components, these oscillator circuits will perform better than almost any other type of clock oscillator, and by almost any criterion of excellence. The suggested circuits are simple, economical, stable, and reliable.

We offer assistance to our customers in selecting suitable off-chip components to work with the on-chip oscillator circuitry. It should be noted, however, that Intel cannot assume the responsibility of writing specifications for the off-chip components of the complete oscillator circuit, nor of guaranteeing the performance of the finished design in production, anymore than a transistor manufacturer, whose data sheets show a number of suggested amplifier circuits, can assume responsibility for the operation, in production, of any of them.

We are often asked why we don't publish a list of required crystal or ceramic resonator specifications, and recommend values for the other off-chip components. This has been done in the past, but sometimes with consequences that were not intended.

Suppose we suggest a maximum crystal resistance of 30 ohms for some given frequency. Then your crystal supplier tells you the 30-ohm crystals are going to cost twice as much as 50-ohm crystals. Fearing that Intel will not "guarantee operation" with 50-ohm crystals, you order the expensive ones. In fact, Intel guarantees only what is embodied within an Intel product. Besides, there is no reason why 50-ohm crystals couldn't be used, if the other off-chip components are suitably adjusted.

Should we recommend values for the other off-chip components? Should we do it for 50-ohm crystals or 30-ohm crystals? With respect to what should we optimize their selection? Should we minimize start-up time or maximize frequency stability? In many applications, neither start-up time nor frequency stability are particularly critical, and our "recommendations" are only restricting your system to unnecessary tolerances. It all depends on the application.

Although we will neither "specify" nor "recommend" specific off-chip components, we do offer assistance in these tasks. Intel application engineers are available to provide whatever technical assistance may be needed or desired by our customers in designing with Intel products.

This Application Note is intended to provide such assistance in the design of oscillator circuits for microcontroller systems. Its purpose is to describe in a practical manner how oscillators work, how crystals and ceramic resonators work (and thus how to spec them), and what the on-chip amplifier looks like electronically and what its operating characteristics are. A BASIC program is provided in Appendix II to assist the designer in determining the effects of changing individual parameters. Suggestions are provided for establishing a pre-production test program.

FEEDBACK OSCILLATORS

Loop Gain

Figure 1 shows an amplifier whose output line goes into some passive network. If the input signal to the amplifier is \( v_1 \), then the output signal from the amplifier is \( v_2 = AV_1 \) and the output signal from the passive network is \( v_3 = \beta v_2 = \beta AV_1 \). Thus \( \beta A \) is the overall gain from terminal 1 to terminal 3.

![Figure 1. Factors in Loop Gain](image)

Now connect terminal 1 to terminal 3, so that the signal path forms a loop: 1 to 2 to 3, which is also 1. Now we have a feedback loop, and the gain factor \( \beta A \) is called the loop gain.

Gain factors are complex numbers. That means they have a magnitude and a phase angle, both of which vary with frequency. When writing a complex number, one must specify both quantities, magnitude and angle. A number whose magnitude is 3, and whose angle is 45 degrees is commonly written this way: \( 3\angle 45^\circ \). The number 1 is, in complex number notation, \( 1\angle 0^\circ \), while \( -1 \) is \( 1\angle 180^\circ \).

By closing the feedback loop in Figure 1, we force the equality

\[ v_1 = \beta A v_1 \]

This equation has two solutions:

1) \( v_1 = 0 \);
2) \( \beta A = 1\angle 0^\circ \).
In a given circuit, either or both of the solutions may be in effect. In the first solution the circuit is quiescent (no output signal). If you’re trying to make an oscillator, a no-signal condition is unacceptable. There are ways to guarantee that the second solution is the one that will be in effect, and that the quiescent condition will be excluded.

How Feedback Oscillators Work

A feedback oscillator amplifies its own noise and feeds it back to itself in exactly the right phase, at the oscillation frequency, to build up and reinforce the desired oscillations. Its ability to do that depends on its loop gain. First, oscillations can occur only at the frequency for which the loop gain has a phase angle of 0 degrees. Second build-up of oscillations will occur only if the loop gain exceeds 1 at the frequency. Build-up continues until nonlinearities in the circuit reduce the average value of the loop gain to exactly 1.

Start-up characteristics depend on the small-signal properties of the circuit, specifically, the small-signal loop gain. Steady-state characteristics of the oscillator depend on the large-signal properties of the circuit, such as the transfer curve (output voltage vs. input voltage) of the amplifier, and the clamping effect of the input protection devices. These things will be discussed more fully further on. First we will look at the basic operation of the particular oscillator circuit, called the “positive reactance” oscillator.

The Positive Reactance Oscillator

Figure 2 shows the configuration of the positive reactance oscillator. The inverting amplifier, working into the impedance of the feedback network, produces an output signal that is nominally 180 degrees out of phase with its input. The feedback network must provide an additional 180 degrees phase shift, such that the overall loop gain has zero (or 360) degrees phase shift at the oscillation frequency.

![Diagram of Positive Reactance Oscillator](image)

**Figure 2. Positive Reactance Oscillator**

In order for the loop gain to have zero phase angle it is necessary that the feedback element \( Z_f \) have a positive reactance. That is, it must be inductive. Then, the frequency at which the phase angle is zero is approximately the frequency at which

\[
X_f = \frac{+1}{\omega C}
\]

where \( X_f \) is the reactance of \( Z_f \) (the total \( Z_f \) being \( R_f + jX_f \)), and \( C \) is the series combination of \( C_{X1} \) and \( C_{X2} \).

\[
C = \frac{C_{X1} C_{X2}}{C_{X1} + C_{X2}}
\]

In other words, \( Z_f \) and \( C \) form a parallel resonant circuit.

If \( Z_f \) is an inductor, then \( X_f = \omega L \), and the frequency at which the loop gain has zero phase is the frequency at which

\[
\omega L = \frac{1}{\omega C}
\]

or

\[
\omega = \frac{1}{\sqrt{LC}}
\]

Normally, \( Z_f \) is not an inductor, but it must still have a positive reactance in order for the circuit to oscillate. There are some piezoelectric devices on the market that show a positive reactance, and provide a more stable oscillation frequency than an inductor will. Quartz crystals can be used where the oscillation frequency is critical, and lower cost ceramic resonators can be used where the frequency is less critical.

When the feedback element is a piezoelectric device, this circuit configuration is called a Pierce oscillator. The advantage of piezoelectric resonators lies in their property of providing a wide range of positive reactance values over a very narrow range of frequencies. The reactance will equal \( 1/\omega C \) at some frequency within this range, so the oscillation frequency will be within the same range. Typically, the width of this range is
only 0.3% of the nominal frequency of a quartz crystal, and about 3% of the nominal frequency of a ceramic resonator. With relatively little design effort, frequency accuracies of 0.03% or better can be obtained with quartz crystals, and 0.3% or better with ceramic resonators.

QUARTZ CRYSTALS

The crystal resonator is a thin slice of quartz sandwiched between two electrodes. Electrically, the device looks pretty much like a 5 or 6 pF capacitor, except that over certain ranges of frequencies the crystal has a positive (i.e., inductive) reactance.

The ranges of positive reactance originate in the piezoelectric property of quartz: Squeezing the crystal generates an internal E-field. The effect is reversible: Applying an AC E-field causes the crystal to vibrate. At certain vibrational frequencies there is a mechanical resonance. As the E-field frequency approaches a frequency of mechanical resonance, the measured reactance of the crystal becomes positive, as shown in Figure 3.

![Figure 3. Crystal Reactance vs. Frequency](image)

Typically there are several ranges of frequencies wherein the reactance of the crystal is positive. Each range corresponds to a different mode of vibration in the crystal. The main resonances are the so-called fundamental response and the third and fifth overtone responses.

The overtone responses shouldn't be confused with the harmonics of the fundamental. They're not harmonics, but different vibrational modes. They're not in general at exact integer multiples of the fundamental frequency. There will also be "spurious" responses, occurring typically a few hundred KHz above each main response.

To assure that an oscillator starts in the desired mode on power-up, something must be done to suppress the loop gain in the undesired frequency ranges. The crystal itself provides some protection against unwanted modes of oscillation; too much resistance in that mode, for example. Additionally, junction capacitances in the amplifying devices tend to reduce the gain at higher frequencies, and thus may discriminate against unwanted modes. In some cases a circuit fix is necessary, such as inserting a trap, a phase shifter, or ferrite beads to kill oscillations in unwanted modes.

Crystal Parameters

Equivalent Circuit

Figure 4 shows an equivalent circuit that is used to represent the crystal for circuit analysis.

The $R_1-L_1-C_1$ branch is called the motivational arm of the crystal. The values of these parameters derive from the mechanical properties of the crystal and are constant for a given mode of vibration. Typical values for various nominal frequencies are shown in Table 1.

![Figure 4. Quartz Crystal: Symbol and Equivalent Circuit](image)

$C_0$ is called the shunt capacitance of the crystal. This is the capacitance of the crystal's electrodes and the mechanical holder. If one were to measure the reactance of the crystal at a frequency far removed from a resonance frequency, it is the reactance of this capacitance that would be measured. It's normally 3 to 7 pF.

<table>
<thead>
<tr>
<th>Frequency MHz</th>
<th>$R_1$ ohms</th>
<th>$L_1$ mH</th>
<th>$C_1$ pF</th>
<th>$C_0$ pF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100</td>
<td>520</td>
<td>0.012</td>
<td>4</td>
</tr>
<tr>
<td>4.608</td>
<td>36</td>
<td>117</td>
<td>0.010</td>
<td>2.9</td>
</tr>
<tr>
<td>11.25</td>
<td>19</td>
<td>8.38</td>
<td>0.024</td>
<td>5.4</td>
</tr>
</tbody>
</table>
The series resonant frequency of the crystal is the frequency at which \( L_1 \) and \( C_1 \) are in resonance. This frequency is given by

\[
f_s = \frac{1}{2\pi \sqrt{L_1 C_1}}
\]

At this frequency the impedance of the crystal is \( R_1 \) in parallel with the reactance of \( C_0 \). For most purposes, this impedance is taken to be just \( R_1 \), since the reactance of \( C_0 \) is so much larger than \( R_1 \).

**Load Capacitance**

A crystal oscillator circuit such as the one shown in Figure 2 (redrawn in Figure 5) operates at the frequency for which the crystal is antiresonant (ie, parallel-resonant) with the total capacitance across the crystal terminals external to the crystal. This total capacitance external to the crystal is called the load capacitance.

As shown in Figure 5, the load capacitance is given by

\[
C_L = \frac{C_{x1} C_{x2}}{C_{x1} + C_{x2}} + C_{\text{stray}}
\]

The crystal manufacturer needs to know the value of \( C_L \) in order to adjust the crystal to the specified frequency.

![Figure 5. Load Capacitance](image)

The adjustment involves putting the crystal in *series* with the specified \( C_L \), and then “trimming” the crystal to obtain resonance of the series combination of the crystal and \( C_L \) at the specified frequency. Because of the high \( Q \) of the crystal, the resonant frequency of the series combination of the crystal and \( C_L \) is the same as the antiresonant frequency of the *parallel* combination of the crystal and \( C_L \). This frequency is given by

\[
f_a = \frac{1}{2\pi \sqrt{L_1 C_1 (C_L + C_0)/(C_1 + C_L + C_0)}}
\]

These frequency formulas are derived (in Appendix A) from the equivalent circuit of the crystal, using the assumptions that the \( Q \) of the crystal is extremely high, and that the circuit external to the crystal has no effect on the frequency other than to provide the load capacitance \( C_L \). The latter assumption is not precisely true, but it is close enough for present purposes.

**“Series” vs. “Parallel” Crystals**

There is no such thing as a “series cut” crystal as opposed to a “parallel cut” crystal. There are different cuts of crystal, having to do with the parameters of its motional arm in various frequency ranges, but there is no special cut for series or parallel operation.

An oscillator is series resonant if the oscillation frequency is \( f_s \) of the crystal. To operate the crystal at \( f_s \), the amplifier has to be noninverting. When buying a crystal for such an oscillator, one does not specify a load capacitance. Rather, one specifies the loading condition as “series.”

If a “series” crystal is put into an oscillator that has an inverting amplifier, it will oscillate in parallel resonance with the load capacitance presented to the crystal by the oscillator circuit, at a frequency slightly above \( f_s \). In fact, at approximately

\[
f_a = f_s \left(1 + \frac{C_1}{2(C_L + C_0)}\right)
\]

This frequency would typically be about 0.02% above \( f_s \).

**Equivalent Series Resistance**

The “series resistance” often listed on quartz crystal data sheets is the real part of the crystal impedance at the crystal’s calibration frequency. This will be \( R_1 \) if the calibration frequency is the series resonant frequency of the crystal. If the crystal is calibrated for parallel resonance with a load capacitance \( C_L \), the equivalent series resistance will be

\[
\text{ESR} = R_1 \left(1 + \frac{C_0}{C_L}\right)^2
\]

The crystal manufacturer measures this resistance at the calibration frequency during the same operation in which the crystal is adjusted to the calibration frequency.
Frequency Tolerance

Frequency tolerance as discussed here is not a requirement on the crystal, but on the complete oscillator. There are two types of frequency tolerances on oscillators: frequency accuracy and frequency stability. Frequency accuracy refers to the oscillator's ability to run at an exact specified frequency. Frequency stability refers to the constancy of the oscillation frequency.

Frequency accuracy requires mainly that the oscillator circuit present to the crystal the same load capacitance that it was adjusted for. Frequency stability requires mainly that the load capacitance be constant.

In most digital applications the accuracy and stability requirements on the oscillator are so wide that it makes very little difference what load capacitance the crystal was adjusted to, or what load capacitance the circuit actually presents to the crystal. For example, if a crystal was calibrated to a load capacitance of 25 pF, and is used in a circuit whose actual load capacitance is 50 pF, the frequency error on that account would be less than 0.01%.

In a positive reactance oscillator, the crystal only needs to be in the intended response mode for the oscillator to satisfy a 0.5% or better frequency tolerance. That's because for any load capacitance the oscillation frequency is certain to be between the crystal's resonant and anti-resonant frequencies.

Phase shifts that take place within the amplifier part of the oscillator will also affect frequency accuracy and stability. These phase shifts can normally be modeled as an "output capacitance" that, in the positive reactance oscillator, parallels $C_X$. The predictability and constancy of this output capacitance over temperature and device sample will be the limiting factor in determining the tolerances that the circuit is capable of holding.

Drive Level

Drive level refers to the power dissipation in the crystal. There are two reasons for specifying it. One is that the parameters in the equivalent circuit are somewhat dependent on the drive level at which the crystal is calibrated. The other is that if the application circuit exceeds the test drive level by too much, the crystal may be damaged. Note that the terms "test drive level" and "rated drive level" both refer to the drive level at which the crystal is calibrated. Normally, in a microcontroller system, neither the frequency tolerances nor the power levels justify much concern for this specification. Some crystal manufacturers don't even require it for microprocessor crystals.

In a positive reactance oscillator, if one assumes the peak voltage across the crystal to be something in the neighborhood of $V_{CC}$, the power dissipation can be approximated as

$$P = 2R_1 \frac{\pi f (C_L + C_0) V_{CC}}{2}$$

This formula is derived in Appendix A. In a 5V system, $P$ rarely evaluates to more than a milliwatt. Crystals with a standard 1 or 2 mW drive level rating can be used in most digital systems.

![Figure 6. Ceramic Resonator Impedance vs. Frequency (Test Data Supplied by NTK Technical Ceramics)](image)

CERAMIC RESONATORS

Ceramic resonators operate on the same basic principles as a quartz crystal. Like quartz crystals, they are piezoelectric, have a reactance versus frequency curve similar to a crystal's, and an equivalent circuit that looks just like a crystal's (with different parameter values, however).

The frequency tolerance of a ceramic resonator is about two orders of magnitude wider than a crystal's, but the ceramic is somewhat cheaper than a crystal. It may be noted for comparison that quartz crystals with relaxed tolerances cost about twice as much as ceramic resonators. For purposes of clocking a microcontroller, the frequency tolerance is often relatively noncritical, and the economic consideration becomes the dominant factor.

Figure 6 shows a graph of impedance magnitude versus frequency for a 3.58 MHz ceramic resonator. (Note that Figure 6 is a graph of $|Z_l|$ versus frequency, where
as Figure 3 is a graph of $X_f$ versus frequency.) A number of spurious responses are apparent in Figure 6. The manufacturers state that spurious responses are more prevalent in the lower frequency resonators (kHz range) than in the higher frequency units (MHz range). For our purposes only the MHz range ceramics need to be considered.

![Figure 7. Ceramic Resonator: Symbol and Equivalent Circuit](image)

Figure 7 shows the symbol and equivalent circuit for the ceramic resonator, both of which are the same as for the crystal. The parameters have different values, however, as listed in Table 2.

<table>
<thead>
<tr>
<th>Frequency (MHz)</th>
<th>$R_1$ (ohms)</th>
<th>$L_1$ (mH)</th>
<th>$C_1$ (pF)</th>
<th>$C_0$ (pF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.58</td>
<td>7</td>
<td>0.113</td>
<td>19.6</td>
<td>140</td>
</tr>
<tr>
<td>6.0</td>
<td>8</td>
<td>0.094</td>
<td>8.3</td>
<td>60</td>
</tr>
<tr>
<td>8.0</td>
<td>7</td>
<td>0.092</td>
<td>4.6</td>
<td>40</td>
</tr>
<tr>
<td>11.0</td>
<td>10</td>
<td>0.057</td>
<td>3.9</td>
<td>30</td>
</tr>
</tbody>
</table>

Note that the motional arm of the ceramic resonator tends to have less resistance than the quartz crystal and also a vastly reduced $L_1/C_1$ ratio. This results in the motional arm having a $Q$ (given by $(1/R_1)(L_1/C_1)$) that is typically two orders of magnitude lower than that of a quartz crystal. The lower $Q$ makes for a faster startup of the oscillator and for a less closely controlled frequency (meaning that circuitry external to the resonator will have more influence on the frequency than with a quartz crystal).

Another major difference is that the shunt capacitance of the ceramic resonator is an order of magnitude higher than $C_0$ of the quartz crystal and more dependent on the frequency of the resonator.

The implications of these differences are not all obvious, but some will be indicated in the section on Oscillator Calculations.

**Specifications for Ceramic Resonators**

Ceramic resonators are easier to specify than quartz crystals. All the vendor wants to know is the desired frequency and the chip you want it to work with. They'll supply the resonators, a circuit diagram showing the positions and values of other external components that may be required and a guarantee that the circuit will work properly at the specified frequency.

**Oscillator Design Considerations**

Designers of microcontroller systems have a number of options to choose from for clocking the system. The main decision is whether to use the "on-chip" oscillator or an external oscillator. If the choice is to use the on-chip oscillator, what kinds of external components are needed to make it operate as advertised? If the choice is to use an external oscillator, what type of oscillator should it be?

The decisions have to be based on both economic and technical requirements. In this section we'll discuss some of the factors that should be considered.

![Figure 8. Using the "On-Chip" Oscillator](image)

**On-Chip Oscillators**

In most cases, the on-chip amplifier with the appropriate external components provides the most economical solution to the clocking problem. Exceptions may arise in severe environments when frequency tolerances are tighter than about 0.01%.

The external components that need to be added are a positive reactance (normally a crystal or ceramic resonator) and the two capacitors $C_{X1}$ and $C_{X2}$, as shown in Figure 8.

**Crystal Specifications**

Specifications for an appropriate crystal are not very critical, unless the frequency is. Any fundamental-mode crystal of medium or better quality can be used.
We are often asked what maximum crystal resistance should be specified. The best answer to this question is the lower the better, but use what's available. The crystal resistance will have some effect on start-up time and steady-state amplitude, but not so much that it can't be compensated for by appropriate selection of the capacitances \( C_X1 \) and \( C_X2 \).

Similar questions are asked about specifications of load capacitance and shunt capacitance. The best advice we can give is to understand what these parameters mean and how they affect the operation of the circuit (that being the purpose of this Application Note), and then decide for yourself if such specifications are meaningful in your application or not. Normally, they're not, unless your frequency tolerances are tighter than about 0.1%.

Part of the problem is that crystal manufacturers are accustomed to talking “ppm” tolerances with radio engineers and simply won’t take your order until you’ve filled out their list of specifications. It will help if you define your actual frequency tolerance requirements, both for yourself and to the crystal manufacturer. Don’t pay for 0.003% crystals if your actual frequency tolerance is 1%.

**Oscillation Frequency**

The oscillation frequency is determined 99.5% by the crystal and up to about 0.5% by the circuit external to the crystal. The on-chip amplifier has little effect on the frequency, which is as it should be, since the amplifier parameters are temperature and process dependent.

The influence of the on-chip amplifier on the frequency is by means of its input and output (pin-to-ground) capacitances, which parallel \( C_X1 \) and \( C_X2 \), and the XTAL1-to-XTAL2 (pin-to-pin) capacitance, which parallels the crystal. The input and pin-to-pin capacitances are about 7 pF each. Internal phase deviations from the nominal 180° can be modeled as an output capacitance of 25 to 30 pF. These deviations from the ideal have less effect in the positive reactance oscillator (with the inverting amplifier) than in a comparable series resonant oscillator (with the noninverting amplifier) for two reasons: first, the effect of the output capacitance is lessened, if not swamped, by the off-chip capacitor; secondly, the positive reactance oscillator is less sensitive, frequency-wise, to such phase errors.

**Selection of \( C_X1 \) and \( C_X2 \)**

Optimal values for the capacitors \( C_X1 \) and \( C_X2 \) depend on whether a quartz crystal or ceramic resonator is being used, and also on application-specific requirements on start-up time and frequency tolerance.

Start-up time is sometimes more critical in microcontroller systems than frequency stability, because of various reset and initialization requirements.

Less commonly, accuracy of the oscillator frequency is also critical, for example, when the oscillator is being used as a time base. As a general rule, fast start-up and stable frequency tend to pull the oscillator design in opposite directions.

Considerations of both start-up time and frequency stability over temperature suggest that \( C_X1 \) and \( C_X2 \) should be about equal and at least 20 pF. (But they don’t have to be either.) Increasing the value of these capacitances above some 40 or 50 pF improves frequency stability. It also tends to increase the start-up time. There is a maximum value (several hundred pF, depending on the value of \( R_1 \) of the quartz or ceramic resonator) above which the oscillator won't start up at all.

If the on-chip amplifier is a simple inverter, such as in the 8051, the user can select values for \( C_X1 \) and \( C_X2 \) between some 20 and 100 pF, depending on whether start-up time or frequency stability is the more critical parameter in a specific application. If the on-chip amplifier is a Schmitt Trigger, such as in the 8048, smaller values of \( C_X1 \) must be used (5 to 30 pF), in order to prevent the oscillator from running in a relaxation mode.

Later sections in this Application Note will discuss the effects of varying \( C_X1 \) and \( C_X2 \) (as well as other parameters), and will have more to say on their selection.

**Placement of Components**

Noise glitches arriving at XTAL1 or XTAL2 pins at the wrong time can cause a miscount in the internal clock-generating circuitry. These kinds of glitches can be produced through capacitive coupling between the oscillator components and PCB traces carrying digital signals with fast rise and fall times. For this reason, the oscillator components should be mounted close to the chip and have short, direct traces to the XTAL1, XTAL2, and VSS pins.

**Clocking Other Chips**

There are times when it would be desirable to use the on-chip oscillator to clock other chips in the system.
This can be done if an appropriate buffer is used. A TTL buffer puts too much load on the on-chip amplifier for reliable start-up. A CMOS buffer (such as the 74HC04) can be used, if it's fast enough and if its VIH and VIL specs are compatible with the available signal amplitudes. Circuits such as shown in Figure 9 might also be considered for these types of applications.

Clock-related signals are available at the TO pin in the MCS-48 products, at ALE in the MCS-48 and MCS-51 lines, and the iACX-96 controllers provide a CLKOUT signal.

### External Oscillators

When technical requirements dictate the use of an external oscillator, the external drive requirements for the microcontroller, as published in the data sheet, must be carefully noted. The logic levels are not in general TTL-compatible. And each controller has its idiosyncracies in this regard. The 8048, for example, requires that both XTAL1 and XTAL2 be driven. The 8051 can be driven that way, but the data sheet suggest the simpler method of grounding XTAL1 and driving XTAL2. For this method, the driving source must be capable of sinking some current when XTAL2 is being driven low.

For the external oscillator itself, there are basically two choices: ready-made and home-grown.
TTL Crystal Clock Oscillator

The HS-100, HS-200, & HS-500 all-metal package series of oscillators are TTL compatible & fit a DIP layout. Standard electrical specifications are shown below. Variations are available for special applications.

Frequency Range: HS-100—3.5 MHz to 30 MHz
  HS-200—225 KHz to 3.5 MHz
  HS-500—25 MHz to 60 MHz

Frequency Tolerance: ±0.1% Overall 0°C–70°C

Hermetically Sealed Package
Mass spectrometer leak rate max.
1 \times 10^{-8} \text{ atmos. cc/sec. of helium}

Output Waveform

\[\text{Output Waveform}\]

INPUT

<table>
<thead>
<tr>
<th></th>
<th>HS-100</th>
<th>HS-200</th>
<th>HS-500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supply Voltage (V_{CC})</td>
<td>5V ±10%</td>
<td>5V ±10%</td>
<td>5V ±10%</td>
</tr>
<tr>
<td>Supply Current (I_{CC}) max.</td>
<td>30 mA</td>
<td>40 mA</td>
<td>85 mA</td>
</tr>
</tbody>
</table>

OUTPUT

<table>
<thead>
<tr>
<th></th>
<th>HS-100</th>
<th>HS-200</th>
<th>HS-500</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V_{OH}) (Logic &quot;1&quot;)</td>
<td>+2.4V min.(^1)</td>
<td>+2.7V min.(^2)</td>
<td>+2.4V min.(^1)</td>
</tr>
<tr>
<td>(V_{OL}) (Logic &quot;0&quot;)</td>
<td>+0.4V max.(^3)</td>
<td>+0.5V max.(^4)</td>
<td>+0.4V max.(^3)</td>
</tr>
<tr>
<td>Symmetry</td>
<td>60/40(^5)</td>
<td>60/40(^5)</td>
<td>55/45(^5)</td>
</tr>
<tr>
<td>(T_{R}, T_{F}) (Rise &amp; Fall Time)</td>
<td>&lt; 10 ns(^6)</td>
<td>&lt; 5 ns(^6)</td>
<td>&lt; 15 ns(^6)</td>
</tr>
<tr>
<td>Output Short Circuit Current</td>
<td>18 mA min.</td>
<td>40 mA min.</td>
<td>18 mA min.</td>
</tr>
<tr>
<td>Output Load</td>
<td>1 to 10 TTL Loads(^7)</td>
<td>1 to 10 TTL Loads(^8)</td>
<td>1 to 10 TTL Loads(^7)</td>
</tr>
</tbody>
</table>

CONDITIONS

\(^1\)I_0 \text{ source} = -400 \mu\text{A max.}
\(^2\)I_0 \text{ source} = -1.0 \mu\text{A max.}
\(^3\)I_0 \text{ sink} = 16.0 \mu\text{A max.}
\(^4\)I_0 \text{ sink} = 20.00 \text{mA max.}
\(^5\)V_{O} = 1.4V
\(^6\)0.4V to 2.4V

Reprinted with the permission of ©Midland-Ross Corporation 1982.
Prepackaged oscillators are available from most crystal manufacturers, and have the advantage that the system designer can treat the oscillator as a black box whose performance is guaranteed by people who carry many years of experience in designing and building oscillators. Figure 10 shows a typical data sheet for some prepackaged oscillators. Oscillators are also available with complementary outputs.

If the oscillator is to drive the microcontroller directly, one will want to make a careful comparison between the external drive requirements in the microcontroller data sheet and the oscillator’s output logic levels and test conditions.

If oscillator stability is less critical than cost, the user may prefer to go with an in-house design. Not without some precautions, however.

It’s easy to design oscillators that work. Almost all of them do work, even if the designer isn’t too clear on why. The key point here is that almost all of them work. The problems begin when the system goes into production, and marginal units commence malfunctioning in the field. Most digital designers, after all, are not very adept at designing oscillators for production.

Oscillator design is somewhat of a black art, with the quality of the finished product being very dependent on the designer’s experience and intuition. For that reason the most important consideration in any design is to have an adequate preproduction test program. Preproduction tests are discussed later in this Application Note. Here we will discuss some of the design options and take a look at some commonly used configurations.

### Gate Oscillators versus Discrete Devices

Digital systems designers are understandably reluctant to get involved with discrete devices and their peculiarities (biasing techniques, etc.). Besides, the component count for these circuits tends to be quite a bit higher than what a digital designer is used to seeing for that amount of functionality. Nevertheless, if there are unusual requirements on the accuracy and stability of the clock frequency, it should be noted that discrete device oscillators can be tailored to suit the exact needs of the application and perfected to a level that would be difficult for a gate oscillator to approach.

In most cases, when an external oscillator is needed, the designer tends to rely on some form of a gate oscillator. A TTL inverter with a resistor connecting the output to the input makes a suitable inverting amplifier. The resistor holds the inverter in the transition region between logical high and low, so that at least for start-up purposes the inverter is a linear amplifier.

The feedback resistance has to be quite low, however, since it must conduct current sourced by the input pin without allowing the DC input voltage to get too far above the DC output voltage. For biasing purposes, the feedback resistance should not exceed a few k-ohms. But shunting the crystal with such a low resistance does not encourage start-up.

Consequently, the configuration in Figure 11A might be suggested. By breaking Rf into two parts and AC-grounding the midpoint, one achieves the DC feedback required to hold the inverter in its active region, but without the negative signal feedback that is in effect telling the circuit not to oscillate. However, this biasing scheme will increase the start-up time, and relaxation-type oscillations are also possible.

A CMOS inverter, such as the 74HC04, might work better in this application, since a larger Rf can be used to hold the inverter in its linear region.

Logic gates tend to have a fairly low output resistance, which destabilizes the oscillator. For that reason a resistor Rx is often added to the feedback network, as shown in Figures 11A and B. At higher frequencies a 20 or 30 pF capacitor is sometimes used in the Rx position, to compensate for some of the internal propagation delay.

Reference 1 contains an excellent discussion of gate oscillators, and a number of design examples.
Fundamental versus Overtone Operation

It's easier to design an oscillator circuit to operate in the resonator's fundamental response mode than to design one for overtone operation. A quartz crystal whose fundamental response mode covers the desired frequency can be obtained up to some 30 MHz. For frequencies above that, the crystal might be used in an overtone mode.

Several problems arise in the design of an overtone oscillator. One is to stop the circuit from oscillating in the fundamental mode, which is what it would really rather do, for a number of reasons, involving both the amplifying device and the crystal. An additional problem with overtone operation is an increased tendency to spurious oscillations. That is because the $R_1$ of various spurious modes is likely to be about the same as $R_1$ of the intended overtone response. It may be necessary, as suggested in Reference 1, to specify a "spurious-to-main-response" resistance ratio to avoid the possibility of trouble.

Overtone oscillators are not to be taken lightly. One would be well advised to consult with an engineer who is knowledgeable in the subject during the design phase of such a circuit.

Series versus Parallel Operation

Series resonant oscillators use noninverting amplifiers. To make a noninverting amplifier out of logic gates requires that two inverters be used, as shown in Figure 12.

This type of circuit tends to be inaccurate and unstable in frequency over variations in temperature and $V_{CC}$. It has a tendency to oscillate at overtones, and to oscillate through $C_0$ of the crystal or some stray capacitance rather than as controlled by the mechanical resonance of the crystal.

The demon in series resonant oscillators is the phase shift in the amplifier. The series resonant oscillator wants more than just a "noninverting" amplifier—it wants a zero phase-shift amplifier. Multistage noninverting amplifiers tend to have a considerably lagging phase shift, such that the crystal reactance must be capacitive in order to bring the total phase shift around the feedback loop back up to 0. In this mode, a "12 MHz" crystal may be running at 8 or 9 MHz. One can put a capacitor in series with the crystal to relieve the crystal of having to produce all of the required phase shift, and bring the oscillation frequency closer to fs. However, to further complicate the situation, the amplifier's phase shift is strongly dependent on frequency, temperature, $V_{CC}$, and device sample.

Positive reactance oscillators ("parallel resonant") use inverting amplifiers. A single logic inverter can be used for the amplifier, as in Figure 11. The amplifier's phase shift is less critical, compared to a series resonant circuit, and since only one inverter is involved there's less phase error anyway. The oscillation frequency is effectively bounded by the resonant and antiresonant frequencies of the crystal itself. In addition, the feedback network includes capacitors that parallel the input and output terminals of the amplifier, thus reducing the effect of unpredictable capacitances at these points.

MORE ABOUT USING THE "ON-CHIP" OSCILLATORS

In this section we will describe the on-chip inverters on selected microcontrollers in some detail, and discuss criteria for selecting components to work with them. Future data sheets will supplement this discussion with updates and information pertinent to the use of each chip's oscillator circuitry.

Oscillator Calculations

Oscillator design, though aided by theory, is still largely an empirical exercise. The circuit is inherently nonlinear, and the normal analysis parameters vary with instantaneous voltage. In addition, when dealing with the on-chip circuitry, we have FETs being used as resistors, resistors being used as interconnects, distributed delays, input protection devices, parasitic junctions, and processing variations.

Consequently, oscillator calculations are never very precise. They can be useful, however, if they will at least indicate the effects of variations in the circuit parameters on start-up time, oscillation frequency, and steady-state amplitude. Start-up time, for example, can be taken as an indication of start-up reliability. If pre-production tests indicate a possible start-up problem, a relatively inexperienced designer can at least be made aware of what parameter may be causing the marginality, and what direction to go in to fix it.
The analysis used here is mathematically straightforward but algebraically intractable. That means it's relatively easy to understand and program into a computer, but it will not yield a neat formula that gives, say, steady-state amplitude as a function of this or that list of parameters. A listing of a BASIC program that implements the analysis will be found in Appendix II.

When the circuit is first powered up, and before the oscillations have commenced (and if the oscillations fail to commence), the oscillator can be treated as a small signal linear amplifier with feedback. In that case, standard small-signal analysis techniques can be used to determine start-up characteristics. The circuit model used in this analysis is shown in Figure 13.

The circuit approximates that there are no high-frequency effects within the amplifier itself, such that its high-frequency behavior is dominated by the load impedance $Z_L$. This is a reasonable approximation for single-stage amplifiers of the type used in 8051-type devices. Then the gain of the amplifier as a function of frequency is

$$A = \frac{A_v Z_L}{Z_L + R_0}$$

The gain of the feedback network is

$$\beta = \frac{Z_i}{Z_i + Z_f}$$

And the loop gain is

$$\beta A = \frac{Z_i}{Z_i + Z_f} \times \frac{A_v Z_L}{Z_L + R_0}$$

The impedances $Z_L$, $Z_f$, and $Z_i$ are defined in Figure 13B.

Figure 14 shows the way the loop gain thus calculated (using typical 8051-type parameters and a 4.608 MHz crystal) varies with frequency. The frequency of interest is the one for which the phase of the loop gain is zero. The accepted criterion for start-up is that the magnitude of the loop gain must exceed unity at this frequency. This is the frequency at which the circuit is in resonance. It corresponds very closely with the antiresonant frequency of the motional arm of the crystal in parallel with $C_L$.

Figure 15 shows the way the loop gain varies with frequency when the parameters of a 3.58 MHz ceramic resonator are used in place of a crystal (the amplifier parameters being typical 8051, as in Figure 14). Note the different frequency scales.
Start-Up Characteristics

It is common, in studies of feedback systems, to examine the behavior of the closed loop gain as a function of complex frequency $s = \sigma + j\omega$; specifically, to determine the location of its poles in the complex plane. A pole is a point on the complex plane where the gain function goes to infinity. Knowledge of its location can be used to predict the response of the system to an input disturbance.

The way that the response function depends on the location of the poles is shown in Figure 16. Poles in the left-half plane cause the response function to take the form of a damped sinusoid. Poles in the right-half plane cause the response function to take the form of an exponentially growing sinusoid. In general,

$$v(t) \sim e^{\alpha t} \sin(\omega t + \theta)$$

where $\alpha$ is the real part of the pole frequency. Thus if the pole is in the right-half plane, $\alpha$ is positive and the sinusoid grows. If the pole is in the left-half plane, $\alpha$ is negative and the sinusoid is damped.

The same type of analysis can usefully be applied to oscillators. In this case, however, rather than trying to ensure that the poles are in the left-half plane, we would seek to ensure that they're in the right-half plane. An exponentially growing sinusoid is exactly what is wanted from an oscillator that has just been powered up.

The gain function of interest in oscillators is $1/(1 - \beta A)$. Its poles are at the complex frequencies where $\beta A = 1 - \angle 0^\circ$, because that value of $\beta A$ causes the gain function to go to infinity. The oscillator will start up if the real part of the pole frequency is positive. More importantly, the rate at which it starts up is indicated by how much greater than 0 the real part of the pole frequency is.

The circuit in Figure 13B can be used to find the pole frequencies of the oscillator gain function. All that needs to be done is evaluate the impedances at complex frequencies $0^\circ + j\omega$ rather than just at $\omega$, and find the value of $0^\circ + j\omega$ for which $\beta A = 1 - \angle 0^\circ$. The larger that value of $\alpha$ is, the faster the oscillator will start up.

Of course, other things besides pole frequencies, things like the VCC rise time, are at work in determining the start-up time. But to the extent that the pole frequencies do affect start-up time, we can obtain results like those in Figures 17 and 18.

To obtain these figures the pole frequencies were computed for various values of capacitance $C_X$ from XTAL1 and XTAL2 to ground (thus $C_X1 = C_X2 = C_X$). Then a “time constant” for start-up was calculated as $T_s = \frac{1}{\sigma}$ where $\sigma$ is the real part of the pole frequency (rad/sec), and this time constant is plotted versus $C_X$. 

![Figure 15. Loop Gain versus Frequency](image1)

![Figure 16. Do You Know Where Your Poles Are Tonight?](image2)
As previously mentioned, start-up time can be taken as an indication of start-up reliability. Start-up problems are normally associated with $C_{X1}$ and $C_{X2}$ being too small or too large for a given resonator. If the parameters of the resonator are known, curves such as in Figure 17 or 18 can be generated to define acceptable ranges of values for these capacitors.

As the oscillations grow in amplitude, they reach a level at which they undergo severe clipping within the amplifier, in effect reducing the amplifier gain. As the amplifier gain decreases, the poles move towards the $j\omega$ axis. In steady-state, the poles are on the $j\omega$ axis and the amplitude of the oscillations is constant.

A short time constant means faster start-up. A long time constant means slow start-up. Observations of actual start-ups are shown in the figures. Figure 17 is for a typical 8051 with a 4.608 MHz crystal supplied by Standard Crystal Corp., and Figure 18 is for a typical 8051 with a 3.58 MHz ceramic resonator supplied by NTK Technical Ceramics, Ltd.

It can be seen in Figure 17 that, for this crystal, values of $C_X$ between 30 and 50 pF minimize start-up time, but that the exact value in this range is not particularly important, even if the start-up time itself is critical.
There is a steady-state analysis in Appendix B that takes some of the complications into account and ignores others. Figure 19 shows the way the steady-state amplitudes thus calculated (using typical 8051 parameters and a 4.608 MHz crystal) vary with equal bulk capacitance placed from XTAL1 and XTAL2 to ground. Experimental results are shown for comparison.

The waveform at XTAL1 is a fairly clean sinusoid. Its negative peak is normally somewhat below zero, at a level which is determined mainly by the input protection circuitry at XTAL1.

The input protection circuitry consists of an ohmic resistor and an enhancement-mode FET with the gate and source connected to ground (VSS), as shown in Figure 20 for the 8051, and in Figure 21 for the 8048. Its function is to limit the positive voltage at the gate of the input FET to the avalanche voltage of the drain junction. If the input pin is driven below VSS, the drain and source of the protection FET interchange roles, so its gate is connected to what is now the drain. In this condition the device resembles a diode with the anode connected to VSS.

There is a parasitic pn junction between the ohmic resistor and the substrate. In the ROM parts (8015, 8048, etc.) the substrate is held at approximately –3V by the on-chip back-bias generator. In the EPROM parts (8751, 8748, etc.) the substrate is connected to VSS.

The effect of the input protection circuitry on the oscillator is that if the XTAL1 signal goes negative, its negative peak is clamped to $-V_{DS}$ of the protection FET in the ROM parts, and to about $-0.5V$ in the EPROM parts. These negative voltages on XTAL1 are in this application self-limiting and nondestructive.

The clamping action does, however, raise the DC level at XTAL1, which in turn tends to reduce the positive peak at XTAL2. The waveform at XTAL2 resembles a sinusoid riding on a DC level, and whose negative peaks are clipped off at zero.

Since it's normally the XTAL2 signal that drives the internal clocking circuitry, the question naturally arises as to how large this signal must be to reliably do its job. In fact, the XTAL2 signal doesn't have to meet the same VIH and VIL specifications that an external oscillator would have to. That's because as long as the oscillator is working, the on-chip amplifier is driving itself through its own 0-to-1 transition region, which is very nearly the same as the 0-to-1 transition region in the internal buffer that follows the oscillator. If some processing variations move the transition level higher or lower, the on-chip amplifier tends to compensate for it by the fact that its own transition level is correspondingly higher or lower. (In the 8096, it's the XTAL1 signal that drives the internal clocking circuitry, but the same concept applies.)

The main concern about the XTAL2 signal amplitude is an indication of the general health of the oscillator. An amplitude of less than about 2.5V peak-to-peak indicates that start-up problems could develop in some units (with low gain) with some crystals (with high $R_1$). The remedy is to either adjust the values of $C_{X1}$ and/or $C_{X2}$ or use a crystal with a lower $R_1$.

The amplitudes at XTAL1 and XTAL2 can be adjusted by changing the ratio of the capacitors from XTAL1 and XTAL2 to ground. Increasing the XTAL2 capacitance, for example, decreases the amplitude at XTAL2 and increases the amplitude at XTAL1 by about the same amount. Decreasing both caps increases both amplitudes.

Figure 19. Calculated and Experimental Steady-State Amplitudes vs. Bulk Capacitance from XTAL1 and XTAL2 to Ground

Steady-State Characteristics

Steady-state analysis is greatly complicated by the fact that we are dealing with large signals and nonlinear circuit response. The circuit parameters vary with instantaneous voltage, and a number of clamping and clipping mechanisms come into play. Analyses that take all these things into account are too complicated to be of general use, and analyses that don't take them into account are too inaccurate to justify the effort.

There is a steady-state analysis in Appendix B that takes some of the complications into account and ignores others. Figure 19 shows the way the steady-state amplitudes thus calculated (using typical 8051 parameters and a 4.608 MHz crystal) vary with equal bulk capacitance placed from XTAL1 and XTAL2 to ground. Experimental results are shown for comparison.

The waveform at XTAL1 is a fairly clean sinusoid. Its negative peak is normally somewhat below zero, at a level which is determined mainly by the input protection circuitry at XTAL1.

The input protection circuitry consists of an ohmic resistor and an enhancement-mode FET with the gate and source connected to ground (VSS), as shown in Figure 20 for the 8051, and in Figure 21 for the 8048. Its function is to limit the positive voltage at the gate of the input FET to the avalanche voltage of the drain junction. If the input pin is driven below VSS, the drain and source of the protection FET interchange roles, so its gate is connected to what is now the drain. In this condition the device resembles a diode with the anode connected to VSS.

There is a parasitic pn junction between the ohmic resistor and the substrate. In the ROM parts (8015, 8048, etc.) the substrate is held at approximately –3V by the on-chip back-bias generator. In the EPROM parts (8751, 8748, etc.) the substrate is connected to VSS.

The effect of the input protection circuitry on the oscillator is that if the XTAL1 signal goes negative, its negative peak is clamped to $-V_{DS}$ of the protection FET in the ROM parts, and to about $-0.5V$ in the EPROM parts. These negative voltages on XTAL1 are in this application self-limiting and nondestructive.

The clamping action does, however, raise the DC level at XTAL1, which in turn tends to reduce the positive peak at XTAL2. The waveform at XTAL2 resembles a sinusoid riding on a DC level, and whose negative peaks are clipped off at zero.

Since it's normally the XTAL2 signal that drives the internal clocking circuitry, the question naturally arises as to how large this signal must be to reliably do its job. In fact, the XTAL2 signal doesn't have to meet the same VIH and VIL specifications that an external oscillator would have to. That's because as long as the oscillator is working, the on-chip amplifier is driving itself through its own 0-to-1 transition region, which is very nearly the same as the 0-to-1 transition region in the internal buffer that follows the oscillator. If some processing variations move the transition level higher or lower, the on-chip amplifier tends to compensate for it by the fact that its own transition level is correspondingly higher or lower. (In the 8096, it's the XTAL1 signal that drives the internal clocking circuitry, but the same concept applies.)

The main concern about the XTAL2 signal amplitude is an indication of the general health of the oscillator. An amplitude of less than about 2.5V peak-to-peak indicates that start-up problems could develop in some units (with low gain) with some crystals (with high $R_1$). The remedy is to either adjust the values of $C_{X1}$ and/or $C_{X2}$ or use a crystal with a lower $R_1$.

The amplitudes at XTAL1 and XTAL2 can be adjusted by changing the ratio of the capacitors from XTAL1 and XTAL2 to ground. Increasing the XTAL2 capacitance, for example, decreases the amplitude at XTAL2 and increases the amplitude at XTAL1 by about the same amount. Decreasing both caps increases both amplitudes.
Pin Capacitance

Internal pin-to-ground and pin-to-pin capacitances at XTAL1 and XTAL2 will have some effect on the oscillator. These capacitances are normally taken to be in the range of 5 to 10 pF, but they are extremely difficult to evaluate. Any measurement of one such capacitance will necessarily include effects from the others. One advantage of the positive reactance oscillator is that the pin-to-ground capacitances are paralleled by external bulk capacitors, so a precise determination of their value is unnecessary. We would suggest that there is little justification for more precision than to assign them a value of 7 pF (XTAL1-to-ground and XTAL1-to-XTAL2). This value is probably not in error by more than 3 or 4 pF.

The XTAL2-to-ground capacitance is not entirely “pin capacitance,” but more like an “equivalent output capacitance” of some 25 to 30 pF, having to include the effect of internal phase delays. This value will vary to some extent with temperature, processing, and frequency.

MCS®-51 Oscillator

The on-chip amplifier on the HMOS MCS-51 family is shown in Figure 20. The drain load and feedback “resistors” are seen to be field-effect transistors. The drain load FET, RD, is typically equivalent to about 1K to 3 K-ohms. As an amplifier, the low frequency voltage gain is normally between -10 and -20, and the output resistance is effectively RD.

The 80151 oscillator is normally used with equal bulk capacitors placed externally from XTAL1 to ground and from XTAL2 to ground. To determine a reasonable value of capacitance to use in these positions, given a crystal of ceramic resonator of known parameters, one can use the BASIC analysis in Appendix II to generate curves such as in Figures 17 and 18. This procedure will define a range of values that will minimize start-up time. We don’t suggest that smaller values be used than those which minimize start-up time. Larger values than those can be used in applications where increased frequency stability is desired, at some sacrifice in start-up time.

Standard Crystal Corp. (Reference 8) studied the use of their crystals with the MCS-51 family using skew sample supplied by Intel. They suggest putting 30 pF capacitors from XTAL1 and XTAL2 to ground, if the crystal is specified as described in Reference 8. They noted that in that configuration and with crystals thus specified, the frequency accuracy was ±0.01% and the frequency stability was ±0.005%, and that a frequency accuracy of ±0.005% could be obtained by substituting a 25 pF fixed cap in parallel with a 5–20 pF trimmer for one of the 30 pF caps.

MCS-51 skew samples have also been supplied to a number of ceramic resonator manufacturers for characterization with their products. These companies should be contacted for application information on their products. In general, however, ceramics tend to want somewhat larger values for CX1 and CX2 than quartz crystals do. As shown in Figure 18, they start up a lot faster that way.

In some application the actual frequency tolerance required is only 1% or so, the user being concerned mainly that the circuit will oscillate. In that case, CX1 and CX2 can be selected rather freely in the range of 20 to 80 pF.

As you can see, “best” values for these components and their tolerances are strongly dependent on the application and its requirements. In any case, their suitability should be verified by environmental testing before the design is submitted to production.

MCS®-48 Oscillator

The NMOS and HMOS MCS-48 oscillator is shown in Figure 21. It differs from the 8051 in that its inverting...
amplifier is a Schmitt Trigger. This configuration was chosen to prevent crosstalk from the TO pin, which is adjacent to the XTAL1 pin.

All Schmitt Trigger circuits exhibit a hysteresis effect, as shown in Figure 22. The hysteresis is what makes it less sensitive to noise. The same hysteresis allows any Schmitt Trigger to be used as a relaxation oscillator. All you have to do is connect a resistor from output to input, and a capacitor from input to ground, and the circuit oscillates in a relaxation mode as follows.

If the Schmitt Trigger output is at a logic high, the capacitor commences charging through the feedback resistor. When the capacitor voltage reaches the upper trigger point (UTP), the Schmitt Trigger output switches to a logic low and the capacitor commences discharging through the same resistor. When the capacitor voltage reaches the lower trigger point (LTP), the Schmitt Trigger output switches to a logic high again, and the sequence repeats. The oscillation frequency is determined by the RC time constant and the hysteresis voltage, UTP-LTP.

The 8048 can oscillate in this mode. It has an internal feedback resistor. All that’s needed is an external capacitor from XTAL1 to ground. In fact, if a smaller external feedback resistor is added, an 8048 system could be designed to run in this mode. Do it at your own risk! This mode of operation is not tested, specified, documented, or encouraged in any way by Intel for the 8048. Future steppings of the device might have a different type of inverting amplifier (one more like the 8051). The CHMOS members of the MCS-48 family do not use a Schmitt Trigger as the inverting amplifier.

Relaxation oscillations in the 8048 must be avoided, and this is the major objective in selecting the off-chip components needed to complete the oscillator circuit.

When an 8048 is powered up, if VCC has a short rise time, the relaxation mode starts first. The frequency is normally about 50 KHz. The resonator mode builds more slowly, but it eventually takes over and dominates the operation of the circuit. This is shown in Figure 23A.

Due to processing variations, some units seem to have a harder time coming out of the relaxation mode, particularly at low temperatures. In some cases the resonator oscillations may fail entirely, and leave the device in the relaxation mode. Most units will stick in the relaxation mode at any temperature if CX1 is larger than about 50 pF. Therefore, CX1 should be chosen with some care, particularly if the system must operate at lower temperatures.

One method that has proven effective in all units to −40°C is to put 5 pF from XTAL1 to ground and 20 pF from XTAL2 to ground. Unfortunately, while this method does discourage the relaxation mode, it is not an optimal choice for the resonator mode. For one thing, it does not swamp the pin capacitance. Also, it makes for a rather high signal level at XTAL1 (8 or 9 volts peak-to-peak).

The question arises as to whether that level of signal at XTAL1 might damage the chip. Not to worry. The negative peaks are self-limiting and nondestructive. The positive peaks could conceivably damage the oxide, but in fact, NMOS chips (eg, 8048) and CHMOS chips (eg, 8048H) are tested to a much higher voltage than that. The technology trend, of course, is to thinner oxides, as the devices shrink in size. For an extra margin of safety, the CHMOS II chips (eg, 8048AH) have an internal diode clamp at XTAL1 to VCC.

In reality, CX1 doesn’t have to be quite so small to avoid relaxation oscillations, if the minimum operating temperature is not −40°C. For less severe temperature requirements, values of capacitance selected in much the same way as for an 8051 can be used. The circuit should be tested, however, at the system’s lowest temperature limit.

Additional security against relaxation oscillations can be obtained by putting a 1M-ohm (or larger) resistor from XTAL1 to VCC. Pulling up the XTAL1 pin this way seems to discourage relaxation oscillations as effectively as any other method (Figure 23B).

Another thing that discourages relaxation oscillations is low VCC. The resonator mode, on the other hand is much less sensitive to VCC. Thus if VCC comes up relatively slowly (several milliseconds rise time), the resonator mode is normally up and running before the relaxation mode starts (in fact, before VCC has even reached operating specs). This is shown in Figure 23C.

A secondary effect of the hysteresis is a shift in the oscillation frequency. At low frequencies, the output signal from an inverter without hysteresis leads (or lags) the input by 180 degrees. The hysteresis in a Schmitt Trigger, however, causes the output to lead the
input by less than 180 degrees (or lag by more than 180 degrees), by an amount that depends on the signal amplitude, as shown in Figure 24. At higher frequencies, there are additional phase shifts due to the various reactances in the circuit, but the phase shift due to the hysteresis is still present. Since the total phase shift in the oscillator's loop gain is necessarily 0 or 360 degrees, it is apparent that as the oscillations build up, the frequency has to change to allow the reactances to compensate for the hysteresis. In normal operation, this additional phase shift due to hysteresis does not exceed a few degrees, and the resulting frequency shift is negligible.

Kyocera, a ceramic resonator manufacturer, studied the use of some of their resonators (at 6.0 MHz, 8.0 MHz, and 11.0 MHz) with the 8049H. Their conclusion as to the value of capacitance to use at XTAL1 and XTAL2 was that 33 pF is appropriate at all three frequencies. One should probably follow the manufacturer's recommendations in this matter, since they will guarantee operation.

Whether one should accept these recommendations and guarantees without further testing is, however, another matter. Not all users have found the recommendations to be without occasional problems. If you run into diffi-

**Figure 23. Relaxation Oscillations in the 8048**
iculties using their recommendations, both Intel and the ceramic resonator manufacturer want to know about it. It is to their interest, and ours, that such problems be resolved.

It will be helpful to build a test jig that will allow the oscillator circuit to be tested independently of the rest of the system. Both start-up and steady-state characteristics should be tested. Figure 25 shows the circuit that

![Image](230659-40)

**A) Inverter Without Hysteresis: Output Leads Input by 180°.**

![Image](230659-41)

**B) Inverter With Hysteresis: Output Leads Input by Less than 180°.**

**Figure 24. Amplitude—Dependent Phase Shift in Schmitt Trigger**

**Preproduction Tests**

An oscillator design should never be considered ready for production until it has proven its ability to function acceptably well under worst-case environmental conditions and with parameters at their worst-case tolerance limits. Unexpected temperature effects in parts that may already be near their tolerance limits can prevent start-up of an oscillator that works perfectly well on the bench. For example, designers often overlook temperature effects in ceramic capacitors. (Some ceramics are down to 50% of their room-temperature values at −20°C and +60°C). The problem here isn’t just one of frequency stability, but also involves start-up time and steady-state amplitude. There may also be temperature effects in the resonator and amplifier.

**Figure 25. Oscillator Test Circuit and Software**

**A) Software for Oscillator Test**

```plaintext
SOURCE
ORG 0000H
JMP START
ORG 0008H ; TIMER 0 INTERRUPT:
CPL T1 ; TOGGLE T1
RETI
ORG 0001BH ; TIMER 1 INTERRUPT:
CPL P1.1 ; TOGGLE CRO TRIGGER
DJNZ P2,$ ; DELAY
CPL P1.0 ; TOGGLE VCC CONTROL
RETI
START: MOV TH1,#0FAH ; TIMER 1 RELOAD VALUE
MOV TL1,#0FAH ; START TL1 AT RELOAD VALUE
MOV TMOD,#16H ; TIMER 1 TO COUNTER, AUTO
; RELOAD
MOV IE,#BAH ; ENABLE TIMER INTERRUPTS
; ONLY
MOV ICON,#50H ; TURN ON BOTH TIMERS
JMP $ ; IDLE

END
```

**B) Oscillator Test Circuit (Shown for 8051 Test)**

![Image](230659-42)
was used to obtain the oscillator start-up photographs in this Application Note. This circuit or a modified version of it would make a convenient test vehicle. The oscillator and its relevant components can be physically separated from the control circuitry, and placed in a temperature chamber.

Start-up should be observed under a variety of conditions, including low VCC and using slow and fast VCC rise times. The oscillator should not be reluctant to start up even when VCC is below its spec value for the rest of the chip. (The rest of the chip may not function, but the oscillator should work.) It should also be verified that start-up occurs when the resonator has more than its upper tolerance limit of series resistance. (Put some resistance in series with the resonator for this test.) The bulk capacitors from XTAL1 and XTAL2 to ground should also be varied to their tolerance limits.

The same circuit, with appropriate changes in the software to lengthen the “on” time, can be used to test the steady-state characteristics of the oscillator, specifically the frequency, frequency stability, and amplitudes at XTAL1 and XTAL2.

As previously noted, the voltage swings at these pins are not critical, but they should be checked at the system’s temperature limits to ensure that they are in good health. Observing these signals necessarily changes them somewhat. Observing the signal at XTAL2 requires that the capacitor at that pin be reduced to account for the oscilloscope probe capacitance. Observing the signal at XTAL1 requires the same consideration, plus a blocking capacitor (switch the oscilloscope input to AC), so as to not disturb the DC level at that pin. Alternatively, a MOSFET buffer such as the one shown in Figure 26 can be used. It should be verified by direct measurement that the ground clip on the scope probe is ohmically connected to the scope chassis (probes are incredibly fragile in this respect), and the observations should be made with the ground clip on the VSS pin, or very close to it. If the probe shield isn’t operational and in use, the observations are worthless.

Frequency checks should be made with only the oscillator circuitry connected to XTAL1 and XTAL2. The ALE frequency can be counted, and the oscillator frequency derived from that. In systems where the frequency tolerance is only “nominal,” the frequency should still be checked to ascertain that the oscillator isn’t running in a spurious resonance or relaxation mode. Switching VCC off and on again repeatedly will help reveal a tendency to go into unwanted modes of oscillation.

The operation of the oscillator should then be verified under actual system running conditions. By this stage one will be able to have some confidence that the basic selection of components for the oscillator itself is suitable, so if the oscillator appears to malfunction in the system the fault is not in the selection of these components.

Troubleshooting Oscillator Problems

The first thing to consider in case of difficulty is that between the test jig and the actual application there may be significant differences in stray capacitances, particularly if the actual application is on a multi-layer board.

Noise glitches, that aren’t present in the test jig but are in the application board, are another possibility. Capacitive coupling between the oscillator circuitry and other signal has already been mentioned as a source of miscounts in the internal clocking circuitry. Inductive coupling is also possible, if there are strong currents nearby. These problems are a function of the PCB layout. Surrounding the oscillator components with “quiet” traces (VCC and ground, for example) will alleviate capacitive coupling to signals that have fast transition times. To minimize inductive coupling, the PCB layout should minimize the areas of the loops formed by the oscillator components. These are the loops that should be checked:

- XTAL1 through the resonator to XTAL2;
- XTAL1 through C_X1 to the VSS pin;
- XTAL2 through C_X2 to the VSS pin.

It is not unusual to find that the grounded ends of C_X1 and C_X2 eventually connect up to the VSS pin only after looping around the farthest ends of the board. Not good.

Finally, it should not be overlooked that software problems sometimes imitate the symptoms of a slow-starting oscillator or incorrect frequency. Never underestimate the perversity of a software problem.
REFERENCES


12. NTK Technical Ceramic Division, NGK Spark Plug Co., Ltd., *NTKK Ceramic Resonator Manual*.
APPENDIX A
QUARTZ AND CERAMIC RESONATOR FORMULAS

Based on the equivalent circuit of the crystal, the impedance of the crystal is

\[
Z_{\text{XTAL}} = \frac{(R_1 + j\omega L_1 + 1/j\omega C_1)(1/j\omega C_0)}{R_1 + j\omega L_1 + 1/j\omega C_1 + 1/j\omega C_0}
\]

After some algebraic manipulation, this calculation can be written in the form

\[
Z_{\text{XTAL}} = \frac{1}{j\omega (C_1 + C_0)} \cdot \frac{1 - \omega^2 L_1 C_1 + j\omega R_1 C_1}{1 - \omega^2 L_1 C_T + j\omega R_1 C_T}
\]

where \( C_T \) is the capacitance of \( C_1 \) in series with \( C_0 \):

\[
C_T = \frac{C_1 C_0}{C_1 + C_0}
\]

The impedance of the crystal in parallel with an external load capacitance \( C_L \) is the same expression, but with \( C_0 + C_L \) substituted for \( C_0 \):

\[
Z_{\text{XTAL}} \parallel C_L = \frac{1}{j\omega (C_1 + C_0 + C_L)} \cdot \frac{1 - \omega^2 L_1 C_1 + j\omega R_1 C_1}{1 - \omega^2 L_1 C_T + j\omega R_1 C_T}
\]

where \( C_T' \) is the capacitance of \( C_1 \) in series with \( C_0 + C_L \):

\[
C_T' = \frac{C_1 (C_0 + C_L)}{C_1 + C_0 + C_L}
\]

The impedance of the crystal in series with the load capacitance is

\[
Z_{\text{XTAL}} + C_L = Z_{\text{XTAL}} + \frac{1}{j\omega C_L}
\]

\[
= \frac{C_L + C_1 + C_0}{j\omega C_L (C_1 + C_0)} \cdot \frac{1 - \omega^2 L_1 C_T' + j\omega R_1 C_T'}{1 - \omega^2 L_1 C_T + j\omega R_1 C_T}
\]

where \( C_T \) and \( C_T' \) are as defined above.

The phase angles of these impedances are readily obtained from the impedance expressions themselves:

\[
\theta_{\text{XTAL}} = \arctan \frac{\omega R_1 C_1}{1 - \omega^2 L_1 C_1}
\]

\[
- \arctan \frac{\omega R_1 C_T'}{1 - \omega^2 L_1 C_T'} - \frac{\pi}{2}
\]

\[
\theta_{\text{XTAL}} + C_L = \arctan \frac{\omega R_1 C_T}{1 - \omega^2 L_1 C_T}
\]

\[
- \arctan \frac{\omega R_1 C_T'}{1 - \omega^2 L_1 C_T'} - \frac{\pi}{2}
\]

The resonant ("series resonant") frequency is the frequency at which the phase angle is zero and the impedance is low. The antiresonant ("parallel resonant") frequency is the frequency at which the phase angle is zero and the impedance is high.

Each of the above \( \theta \)-expressions contains two \( \arctan \) functions. Setting the denominator of the argument of the first \( \arctan \) function to zero gives (approximately) the "series resonant" frequency for that configuration. Setting the denominator of the argument of the second \( \arctan \) function to zero gives (approximately) the "parallel resonant" frequency for that configuration.

For example, the resonant frequency of the crystal is the frequency at which

\[
1 - \omega^2 L_1 C_1 = 0
\]

Thus

\[
\omega_s = \frac{1}{\sqrt{L_1 C_1}}
\]

or

\[
f_s = \frac{1}{2\pi \sqrt{L_1 C_1}}
\]
It will be noted that the series resonant frequency of the “XTAL + CL” configuration (crystal in series with CL) is the same as the parallel resonant frequency of the “XTAL||CL” configuration (crystal in parallel with CL). This is the frequency at which

\[ 1 - \omega^2 L_1 C_T' = 0 \]

Thus \[ \omega_a = \frac{1}{\sqrt{L_1 C_T}} \]

or \[ f_a = \frac{1}{2\pi \sqrt{L_1 C_T}} \]

This fact is used by crystal manufacturers in the process of calibrating a crystal to a specified load capacitance.

By subtracting the resonant frequency of the crystal from its antiresonant frequency, one can calculate the range of frequencies over which the crystal reactance is positive:

\[ f_a - f_s = f_a \left( \frac{C_1}{2C_0} \right) \]

Given typical values for C_1 and C_0, this range can hardly exceed 0.5% of f_s. Unless the inverting amplifier in the positive reactance oscillator is doing something very strange indeed, the oscillation frequency is bound to be accurate to that percentage whether the crystal was calibrated for series operation or to any unspecified load capacitance.

### Equivalent Series Resistance

ESR is the real part of Z_{XTAL} at the oscillation frequency. The oscillation frequency is the parallel resonant frequency of the “XTAL||CL” configuration (which is the same as the series resonant frequency of the “XTAL + CL” configuration). Substituting this frequency into the Z_{XTAL} expression yields, after some algebraic manipulation,

\[ ESR = R_1 \left( \frac{C_0 + C_L}{C_L} \right)^2 \]

\[ = R_1 \left( 1 + \frac{C_0}{C_L} \right)^2 \]

### Drive Level

The power dissipated by the crystal is \( I_1^2 R_1 \), where \( I_1 \) is the RMS current in the motional arm of the crystal. This current is given by \( V_x / |Z_1| \), where \( V_x \) is the RMS voltage across the crystal, and \( |Z_1| \) is the magnitude of the impedance of the motional arm. At the oscillation frequency, the motional arm is a positive (inductive) reactance in parallel resonance with \((C_0 + C_L)\). Therefore \(|Z_1|\) is approximately equal to the magnitude of the reactance of \((C_0 + C_L)\):

\[ |Z_1| = \frac{1}{2\pi f (C_0 + C_L)} \]

where \( f \) is the oscillation frequency. Then,

\[ P = I_1^2 R_1 = \left( \frac{V_x}{|Z_1|} \right)^2 R_1 \]

\[ = 2\pi f (C_0 + C_L) \frac{V_x}{\sqrt{2}} R_1 \]

The waveform of the voltage across the crystal (XTAL1 to XTAL2) is approximately sinusoidal. If its peak value is VCC, then \( V_x \) is VCC/\sqrt{2}. Therefore,

\[ P = 2R_1 \left[ \pi f (C_0 + C_L) \text{VCC} \right]^2 \]
APPENDIX B
OSCILLATOR ANALYSIS PROGRAM

The program is written in BASIC. BASIC is excruciatingly slow, but it has some advantages. For one thing, more people know BASIC than FORTRAN. In addition, a BASIC program is easy to develop, modify, and “fiddle around” with. Another important advantage is that a BASIC program can run on practically any small computer system.

Its slowness is a problem, however. For example, the routine which calculates the “start-up time constant” discussed in the text may take several hours to complete. A person who finds this program useful may prefer to convert it to FORTRAN, if the facilities are available.

Limitations of the Program

The program was developed with specific reference to 8051-type oscillator circuitry. That means the on-chip amplifier is a simple inverter, and not a Schmitt Trigger. The 8096, the 80C51, the 80C48 and 80C49 all have simple inverters. The 8096 oscillator is almost identical to the 8051, differing mainly in the input protection circuitry. The CHMOS amplifiers have somewhat different parameters (higher gain, for example), and different transition levels than the 8096.

The MCS-48 family is specifically included in the program only to the extent that the input-output curve used in the steady-state analysis is that of a Schmitt Trigger, if the user identifies the device under analysis as an MCS-48 device. The analysis does not include the voltage dependent phase shift of the Schmitt Trigger.

The clamping action of the input protection circuitry is important in determining the steady-state amplitudes. The steady-state routine accounts for it by setting the negative peak of the XTAL1 signal at a level which depends on the amplitude of the XTAL1 signal in accordance with experimental observations. It’s an exercise in curve-fitting. A user may find a different type of curve works better. Later stepping of the chips may behave differently in this respect, having somewhat different types of input protection circuitry.

It should be noted that the analysis ignores a number of important items, such as high-frequency effects in the on-chip circuitry. These effects are difficult to predict, and are no doubt dependent on temperature, frequency, and device sample. However, they can be simulated to a reasonable degree by adding an “output capacitance” of about 20 pF to the circuit model (i.e., in parallel with C2) as described below.

Notes on Using the Program

The program asks the user to input values for various circuit parameters. First the crystal (or ceramic resonator) parameters are asked for. These are R1, L1, C1, and C0. The manufacturer can supply these values for selected samples. To obtain any kind of correlation between calculation and experiment, the values of these parameters must be known for the specific sample in the test circuit. The value that should be entered for C0 is the C0 of the crystal itself plus an estimated 7 pF to account for the XTAL1-to-XTAL2 pin capacitance, plus any other stray capacitance paralleling the crystal that the user may feel is significant enough to be included.

Then the program asks for the values of the XTAL1-to-ground and XTAL2-to-ground capacitances. For CXTAL1, enter the value of the externally connected bulk capacitor plus an estimated 7 pF for pin capacitance. For CXTAL2, enter the value of the externally connected bulk capacitor plus an estimated 7 pF for pin capacitance plus about 20 pF to simulate high-frequency roll-off and phase shifts in the on-chip circuitry.

Next the program asks for values for the small-signal parameters of the on-chip amplifier. Typically, for the 8051/8751,

\[
\begin{align*}
\text{Amplifier Gain Magnitude} & = 15 \\
\text{Feedback Resistance} & = 2300 \text{ K}\Omega \\
\text{Output Resistance} & = 2 \text{ K}\Omega
\end{align*}
\]

The same values can be used for MCS-48 (NMOS and HMOS) devices, but they are difficult to verify, because the Schmitt Trigger does not lend itself to small-signal measurements.
100 DEFDBL C.D.F.G.L.P.R.S.X
200 REM **********************************************************************
300 REM ************************************************************************
400 REM FUNCTIONS
500 REM
600 REM
700 REM
800 REM FNZM(R.X) = MAGNITUDE OF A COMPLEX NUMBER. IR+JX:
900 DEF FNZM(R.X) = SQR(R^2+X^2)
1000 REM
1100 REM FNZP(R.X) = ANGLE OF A COMPLEX NUMBER
1200 REM = 180/P I*ARCTAN(X/R) + 180 IF R>0
1300 REM = 180/P I*ARCTAN(X/R) - 180 IF R<0 AND X>0
1400 REM = 180/P I*ATN(X/R) - (SGN(R)-1)*SGN(X)*90
1500 REM
1600 REM
1700 REM INDUCTIVE IMPEDANCE AT COMPLEX FREQUENCY S+JF (HZ)
1800 REM Z = 2*PI*S*L + J2*PI*F*L
1900 REM FNRL(S.L) + JFNXL(F.L)
2000 REM FNRL(SL.LL) = 2*PI*S*L*LL
2100 REM FNXL(FL.LL) = 2*PI*L*X*LL
2200 REM
2300 REM CAPACITIVE IMPEDANCE AT COMPLEX FREQUENCY S+JF (HZ)
2400 REM Z = 1/(2*PI*iS+JF)*C
2500 REM = S/(2*PI*(S^2+F^2)*C) + Jf-F)/[2*PI*(S+A^2)C]
2600 REM FNRC(SC.FC.CC) = SC/(2*PI*(SC^2+FC^2)*CC)
2700 REM FNXC(SC.FC.CC) = FC/(2*PI*(SC^2+FC^2)*CC)
2800 REM
2900 REM RATIO OF TWO COMPLEX NUMBERS
3000 REM RA+JXA RA*RB+XAXB XAXB-RA*XAXB
3100 REM = FNRR(RA.XA.RB.XB) + JFNXR(RA.XA.RB.XB)
3200 REM FNRR(RA.XA.RB.XB) = (RA*RD+XA*XD)/(RD^2+XD^2)
3300 REM FNXR(RA.XA.RB.XB) = (XA*RO-XB*RA)/(RB^2+XB^2)
3400 REM
3500 REM PRODUCT OF TWO COMPLEX NUMBERS
3600 REM (RA+JXA)*(RO+JXB) = RA*RO-XA*XB + J(XA*RO+RA*XB)
3700 REM FNRC(SC.FC.CC) = SC/(2*PI*(SC^2+FC^2)*CC)
3800 REM FNRC(SC.FC.CC) = SC/(2*PI*(SC^2+FC^2)*CC)
3900 REM
4000 REM PARALLEL IMPEDANCES
4100 REM (RA+JXA)*C(RB+JXB) = (RA+JXA)*(RB+JXB)
4200 REM FNRR(RA.XA.RB.XB) = RA*RB - XAXB
4300 REM FNXR(RA.XA.RB.XB) = XAXB + RA*RB
4400 REM
4500 REM BEGIN COMPUTATIONS
4600 LET PI = 3.141592654
4700 REM DEFINE CIRCUIT PARAMETERS
4800 REM ESTABLISH NOMINAL RESONANT AND ANTI-RESONANT CRYSTAL FREQUENCIES
4900 REM FS = FIX(1/(2*PI*SQR(L1*C1)))
5000 REM FA = FIX(1/(2*PI*SQR(L1*C1)*C1*C0)))
5100 REM
5200 REM
5300 REM PRINT "XTAL IS SERIES RESONANT AT", FS, "Hz"
5400 REM PRINT "PARALLEL RESONANT AT", FA, "Hz"
5500 REM
5600 REM
5700 REM BEGIN COMPUTATIONS
5800 REM
5900 REM
6000 REM
6100 REM
6200 REM LET PI = 3.141592654
6300 REM
6400 REM
6500 REM
6600 REM
6700 REM ESTABLISH NOMINAL RESONANT AND ANTI-RESONANT CRYSTAL FREQUENCIES
6800 FS = FIX(1/(2*PI*SQR(L1*C1)))
6900 FA = FIX(1/(2*PI*SQR(L1*C1)*C1*C0)))
7000 PRINT
7100 PRINT "XTAL IS SERIES RESONANT AT", FS, "Hz"
7200 PRINT "PARALLEL RESONANT AT", FA, "Hz"
7300 PRINT
7400 PRINT "SELECT 1 LIST PARAMETERS"
7500 PRINT "2 CIRCUIT ANALYSIS"
7600 PRINT "3 OSCILLATION FREQUENCY"
7700 PRINT "4 START-UP TIME CONSTANT"
7800 PRINT "5 STEADY-STATE ANALYSIS"
7900 PRINT
8000 INPUT N
8100 IF N=1 THEN PRINT ELSE B600
8200 REM
8300 REM ----------------------------------- LIST PARAMETERS -----------------------------------
8400 GOSUB 17100
8500 GOTO 6800
8600 IF N=2 THEN PRINT ELSE 8600
8700 REM
8800 REM ----------------------------------- CIRCUIT ANALYSIS -----------------------------------
8900 PRINT " FREQUENCY 3.\F TYPE (S)/(F) "
9000 INPUT SB.FQ
9100 GOSUB 20200
9200 GOSUB 26600
9300 GOTO 6E100
9400 IF N=3 THEN PRINT ELSE 11000
9500 REM
9600 REM ----------------------------------- OSCILLATION FREQUENCY -----------------------------------
9700 CL = C1*C/(C1+C2) + CO
9800 SF = FIX((1/(2*PI*SQR(L1*C1+C1*C2))))
9900 SQ = 0
10000 DF = FIX((10^INT(LOG(FQ-BS)/LOG(1.1-2)+5))
10100 DS = 0
10200 RETURN
10300 GOSUB 9700
10400 GOSUB 30300
10500 PRINT
10600 PRINT
10700 PRINT " FREQUENCY AT WHICH LOOP GAIN HAS ZERO PHASE ANGLE. "
10800 GOSUB 26600
10900 GOTO 6800
11000 IF N=4 THEN PRINT ELSE 12200
11100 REM
11200 REM ----------------------------------- START-UP TIME CONSTANT -----------------------------------
11300 PRINT " THIS WILL TAKE SOME TIME "
11400 GOSUB 9700
11500 GOSUB 37700
11600 PRINT
11700 PRINT
11800 PRINT " FREQUENCY AT WHICH LOOP GAIN = 1 AT 0 DEGREES. "
11900 GOSUB 26600
12000 PRINT : PRINT " THIS YIELDS A START-UP TIME CONSTANT OF : CSNG((1000000/(2*PI*SQ))) MICROSECS"
12100 GOTO 6800
12200 IF N=5 THEN PRINT ELSE 7300
12300 REM
12400 REM ----------------------------------- STEADY-STATE ANALYSIS -----------------------------------
12500 PRINT " STEADY-STATE ANALYSIS"
12600 PRINT
12700 PRINT " SELECT: 1. B03/8031 "
12800 PRINT " 2. B750 "
12900 PRINT " 3. B239/B040/B048/B049 "
13000 PRINT " 4. B749/B749 "
13100 INPUT ICX
13200 IF ICX<1 OR ICX>4 THEN 12600
13300 GOSUB 46900
13400 GOTO 7300
13500 REM SUBROUTINE BELOW DEFINES INPUT-OUTPUT CURVE OF OSCILLATOR CKT
13600 IF ICX=2 AND VO<5 AND VI=2 THEN RETURN
13700 VO = 10*VI + 9
13800 IF VO>5 THEN VO = 5
13900 IF VO<2 THEN VO = .2
14000 IF ICX>2 AND VO<2 THEN VO = 5
14100 RETURN
14200 REM
14300 REM **************************************************************
14400 REM DEFINE CIRCUIT PARAMETERS
14500 REM
14600 REM
14700 INPUT = R1 (OHMS); R1
14800 INPUT = L1 (HENRY); L1
14900 INPUT = C1 (PF); X
15000 C1 = X*1E-12
15100 INPUT = C0 (PF); X
15200 C0 = X*1E-12
15300 INPUT = CXTAL1 (PF); X
15400 CX = X*1E-12
15500 INPUT = CXTAL2 (PF); X
15600 CY = X*1E-12

230859-45
15700 INPUT "GAIN FACTOR MAGNITUDE": AV#
15800 INPUT "AMP FEEDBACK RESISTANCE (K-OHMS)": X
15900 RX = X*1000
16000 INPUT "AMP OUTPUT RESISTANCE (K-OHMS)": X
16100 RO = X*1000
16200 REM
16300 REM  LIST CURRENT PARAMETER VALUES
16500 QOSUB 17100
16600 RETURN
16700 REM
16800 REM
16900 REM  ******************************************************
17000 REM
17100 REM  LIST CURRENT PARAMETER VALUES
17200 REM
17300 PRINT
17400 PRINT "CURRENT PARAMETER VALUES
17500 PRINT 1 RI = ", RI, " OHMS"
17600 PRINT 2 LI = ", CSNG(L1), " HENRY"
17700 PRINT 3 CI = ", CSNG(C1*1E+12), " PF"
17800 PRINT 4 CO = ", CSNG(CD*1E+12), " PF"
17900 PRINT 5 CXTAL1 = ", CSNG(CX*1E+12), " PF"
17900 PRINT 6 CXTAL2 = ", CSNG(CY*1E+12), " PF"
18000 PRINT 7 AMPLIFIER GAIN MAGNITUDE = ", AV#
18100 PRINT 8 FEEDBACK RESISTANCE = ", CSNG(RX* 001), " K-OHMS"
18200 PRINT 9 OUTPUT RESISTANCE = ", CSNG(RO* 001), " K-OHMS"
18300 PRINT
18400 PRINT "TO CHANGE A PARAMETER VALUE, TYPE (PARAM NO),(NEW VALUE)"
18500 PRINT "OTHERWISE, TYPE 0,0"
18600 INPUT N$,X
18700 IF N$="0" THEN RETURN
18800 IF N$="1" THEN RI = X
18900 IF N$="2" THEN LI = X
19000 IF N$="3" THEN CI = X*1E-12
19100 IF N$="4" THEN CO = X*1E-12
19200 IF N$="5" THEN CX = X*1E-12
19300 IF N$="6" THEN CY = X*1E-12
19400 IF N$="7" THEN AV# = X
19500 IF N$="8" THEN RX = X*1000
19600 IF N$="9" THEN RO = X*1000
19700 QOTO 17400
19800 REM
19900 REM
20000 REM  ******************************************************
20100 REM
20200 REM
20300 REM  CIRCUIT ANALYSIS
20400 REM
20500 REM
20600 REM  This routine calculates the loop gain at complex frequency SQ+JFQ.
20700 REM
20800 REM
20900 REM
21000 REM
21100 REM
21200 REM
21300 REM
21400 REM
21500 REM
21600 REM
21700 REM
21800 REM
21900 REM
22000 REM
22100 REM
22200 REM
22300 REM
22400 REM
22500 REM
22600 REM
22700 REM
22800 REM
22900 REM
23000 REM
23100 REM
23200 REM
23300 REM
23400 REM

******.*~****~*4*4*.~*****.***.*****************

230659-46
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>23500</td>
<td>REM</td>
</tr>
<tr>
<td>23600</td>
<td>BRM = FNRM(RI, XI, (RI+RF), (XI+XF))</td>
</tr>
<tr>
<td>23700</td>
<td>DIN = FNRX(RI, XI, (RI+RF), (XI+XF))</td>
</tr>
<tr>
<td>23800</td>
<td>REM</td>
</tr>
<tr>
<td>23900</td>
<td>REM 7 Amplifier gain in magnitude/phase form AR+jAI = A at AP degrees</td>
</tr>
<tr>
<td>24000</td>
<td>REM</td>
</tr>
<tr>
<td>24100</td>
<td>A = FNZI(A, A, A, A)</td>
</tr>
<tr>
<td>24200</td>
<td>AP = FNZP(A, A, A, A)</td>
</tr>
<tr>
<td>24300</td>
<td>REM</td>
</tr>
<tr>
<td>24400</td>
<td>REM 0 (beta) in magnitude/phase form BR+jBI = 0 at BP degrees</td>
</tr>
<tr>
<td>24500</td>
<td>REM</td>
</tr>
<tr>
<td>24600</td>
<td>B = FNZI(B, B, B, B)</td>
</tr>
<tr>
<td>24700</td>
<td>BP = FNZP(B, B, B, B)</td>
</tr>
<tr>
<td>24800</td>
<td>REM</td>
</tr>
<tr>
<td>24900</td>
<td>REM 9 Loop gain G = (BR+jBI)*(AR+jAI)</td>
</tr>
<tr>
<td>25000</td>
<td>REM</td>
</tr>
<tr>
<td>25100</td>
<td>REM</td>
</tr>
<tr>
<td>25200</td>
<td>GR = FNRM(A, A, A, A, B, B)</td>
</tr>
<tr>
<td>25300</td>
<td>GI = FNXM(A, A, A, A, B, B)</td>
</tr>
<tr>
<td>25400</td>
<td>REM</td>
</tr>
<tr>
<td>25500</td>
<td>REM 10 Loop gain in magnitude/phase form GR+jGI = AL at AQ degrees</td>
</tr>
<tr>
<td>25600</td>
<td>REM</td>
</tr>
<tr>
<td>25700</td>
<td>AL = FNZM(GR, GI)</td>
</tr>
<tr>
<td>25800</td>
<td>AQ = FNZM(GR, GI)</td>
</tr>
<tr>
<td>25900</td>
<td>RETURN</td>
</tr>
<tr>
<td>26000</td>
<td>REM</td>
</tr>
<tr>
<td>26100</td>
<td>REM *************************************************************************</td>
</tr>
<tr>
<td>26200</td>
<td>REM</td>
</tr>
<tr>
<td>26400</td>
<td>REM PRINT CIRCUIT ANALYSIS RESULTS</td>
</tr>
<tr>
<td>26500</td>
<td>REM</td>
</tr>
<tr>
<td>26600</td>
<td>PRINT</td>
</tr>
<tr>
<td>26700</td>
<td>PRINT &quot; FREQUENCY = &quot;, SG, &quot; + J&quot;, FG, &quot; HZ &quot;</td>
</tr>
<tr>
<td>26800</td>
<td>PRINT &quot; XTAL IMPEDANCE = &quot;, FNZM(RE, XE), &quot; OHMS AT &quot;, FNZM(RE, XE), &quot; DEGREES &quot;</td>
</tr>
<tr>
<td>26900</td>
<td>PRINT (RE = &quot;CSNG(RE), &quot; OHMS)</td>
</tr>
<tr>
<td>27000</td>
<td>PRINT &quot; (XE = &quot;CSNG(XE), &quot; OHMS)</td>
</tr>
<tr>
<td>27100</td>
<td>PRINT &quot; LOAD IMPEDANCE = &quot;, FNZM(RL, XL), &quot; OHMS AT &quot;, FNZM(RL, XL), &quot; DEGREES &quot;</td>
</tr>
<tr>
<td>27200</td>
<td>PRINT &quot; AMPLIFIER GAIN = &quot;, A, &quot; AT &quot;, AP, &quot; DEGREES &quot;</td>
</tr>
<tr>
<td>27300</td>
<td>PRINT &quot; FEEDBACK RATIO = &quot;, B, &quot; AT &quot;, BP, &quot; DEGREES &quot;</td>
</tr>
<tr>
<td>27400</td>
<td>PRINT &quot; LOOP GAIN = &quot;, AL, &quot; AT &quot;, AQ, &quot; DEGREES &quot;</td>
</tr>
<tr>
<td>27500</td>
<td>RETURN</td>
</tr>
<tr>
<td>27600</td>
<td>REM</td>
</tr>
<tr>
<td>27700</td>
<td>REM</td>
</tr>
<tr>
<td>27800</td>
<td>REM *************************************************************************</td>
</tr>
<tr>
<td>27900</td>
<td>REM SEARCH FOR FREQUENCY (S+JF)</td>
</tr>
<tr>
<td>28100</td>
<td>REM AT WHICH LOOP GAIN HAS ZERO PHASE ANGLE</td>
</tr>
<tr>
<td>28200</td>
<td>REM</td>
</tr>
<tr>
<td>28300</td>
<td>REM This routine searches for the frequency at which the imaginary part of the loop gain is zero. The algorithm is as follows:</td>
</tr>
<tr>
<td>28400</td>
<td>REM of the loop gain is zero. The algorithm is as follows:</td>
</tr>
<tr>
<td>28500</td>
<td>REM 1. Calculate the sign of the imaginary part of the loop gain (GI),</td>
</tr>
<tr>
<td>28600</td>
<td>REM 2. Increment the frequency</td>
</tr>
<tr>
<td>28700</td>
<td>REM 3. Calculate the sign of GI at the incremented frequency</td>
</tr>
<tr>
<td>28800</td>
<td>REM 4. If the sign of GI has not changed, go back to 2</td>
</tr>
<tr>
<td>28900</td>
<td>REM 5. If the sign of GI has changed, and this frequency is within 1Hz of the previous sign-change, exit the routine</td>
</tr>
<tr>
<td>29100</td>
<td>REM 6. Otherwise, divide the frequency increment by -10</td>
</tr>
<tr>
<td>29200</td>
<td>REM 7. Go back to 2</td>
</tr>
<tr>
<td>29300</td>
<td>REM The routine is entered with the starting frequency SG+JFQ and starting increment DS+JDF already defined by the calling program.</td>
</tr>
<tr>
<td>29400</td>
<td>REM In actual use either DS or DF is zero, so the routine searches for</td>
</tr>
<tr>
<td>29500</td>
<td>REM a GI=0 point by incrementing either SQ or FQ while holding the other</td>
</tr>
<tr>
<td>29700</td>
<td>REM constant. It returns control to the calling program with the</td>
</tr>
<tr>
<td>29800</td>
<td>REM incremented part of the frequency being within 1Hz of the actual</td>
</tr>
<tr>
<td>29900</td>
<td>REM GI=0 point</td>
</tr>
<tr>
<td>30000</td>
<td>REM</td>
</tr>
<tr>
<td>30100</td>
<td>REM 1. CALCULATE THE SIGN OF THE IMAGINARY PART OF THE LOOP GAIN (GI)</td>
</tr>
<tr>
<td>30200</td>
<td>REM</td>
</tr>
<tr>
<td>30300</td>
<td>GOSUB 20200</td>
</tr>
<tr>
<td>30400</td>
<td>GOSUB 26600</td>
</tr>
<tr>
<td>30500</td>
<td>IF GI=0 THEN RETURN</td>
</tr>
<tr>
<td>30600</td>
<td>5X = INT(SNGL(GI))</td>
</tr>
<tr>
<td>30700</td>
<td>IF X==1 THEN DS = -DS</td>
</tr>
<tr>
<td>30800</td>
<td>REM (REVERSAL OF DS FOR GI=0 IS FOR THE POLE-SEARCH ROUTINE )</td>
</tr>
<tr>
<td>30900</td>
<td>REM</td>
</tr>
<tr>
<td>31000</td>
<td>REM 2 INCREMENT THE FREQUENCY</td>
</tr>
<tr>
<td>31100</td>
<td>REM</td>
</tr>
<tr>
<td>31200</td>
<td>SP = SQ</td>
</tr>
</tbody>
</table>

230659-47
31200 FP = FQ  
31400 SQ = SQ + DS  
31500 FQ = FQ + DF  
31600 REM 3 CALCULATE THE SIGN OF GI AT THE INCREMENTS FREQUENCY  
31800 REM 39000 FCl = FCl  
31900 GOSUB 20200  
32000 GOSUB 26600  
32100 IF INT(SGN(GL)) = 0 THEN RETURN  
32200 REM 32300 4 IF THE SIGN OF GI HAS NOT CHANGED, GO BACK TO 2  
32400 REM 32500 IF SIG*INT(SGN(GL)) = 0 THEN PRINT ELSE 31400  
32600 REM 32700 SQ = -SQ  
32800 REM 5 IF THE SIGN OF GI HAS CHANGED, AND IF THIS FREQUENCY IS WITHIN  
32900 REM 1Hz OF THE PREVIOUS SIGN-CHANGE, AND IF GI IS NEGATIVE, THEN  
33000 REM EXIT THE ROUTINE. THE ADDITIONAL REQUIREMENT FOR NEGATIVE GI  
33100 REM IS FOR THE POLE-SEARCH ROUTINE  
33200 REM 33300 IF ABS(SP*SQ<1 AND abs(FP-FG)<:1 AND SQ<1 THEN RETURN  
33400 REM 33500 REM 6. DIVIDE THE FREQUENCY INCREMENT BY -10  
33600 REM 33700 DS = -DS/10#  
33800 DF = -DF/10#  
33900 REM 34000 REM 7. GO BACK TO 2  
34100 REM 34200 GOTO 31200  
34300 REM 34400 REM SEARCH FOR POLE FREQUENCY  
34700 REM 34900 REM This routine searches for the frequency at which the loop gain = 1  
35000 REM at 0 degrees. That frequency is the pole frequency of the closed-loop  
35100 REM loop gain function. The pole frequency is a complex number, SQ+JFQ  
35200 REM (Hz). Oscillator start-up ensues if SQ<0. The algorithm is based on  
35300 REM the calculated behavior of the phase angle of the loop gain in the  
35400 REM region of interest on the complex plane. The locus of points of zero  
35500 REM phase angle crosses the j-axis at the oscillation frequency and at  
35600 REM some higher frequency. In between these two crossings of the j-axis,  
35700 REM the locus lies in Quadrant I of the complex plane, forming an  
35800 REM approximate parabola which opens to the left. The basic plan is to  
35900 REM follow the locus from where it crosses the j-axis at the oscillation  
36000 REM frequency, into Quadrant I, and find the point on that locus where  
36100 REM the loop gain has a magnitude of 1. The algorithm is as follows:  
36200 REM 1. Find the oscillation frequency. O+JFQ  
36300 REM 2. At this frequency calculate the sign of (AL-1) (AL = magnitude  
36400 REM of loop gain)  
36500 REM 3. Increment FQ  
36600 REM 4. For this value of FQ, find the value of SQ for which the loop  
36700 REM gain has zero phase  
36800 REM 5. For this value of SQ+JFQ, calculate the sign of (AL-1)  
36900 REM 6. If the sign of (AL-1) has not changed, go back to 3  
37000 REM 7. If the sign of (AL-1) has changed, and this value of FQ is  
37100 REM within 1Hz of the previous sign-change, exit the routine  
37200 REM 8. Otherwise, divide the FQ increment by -10  
37300 REM 9. Go back to 3  
37400 REM 37500 REM 1 FIND THE OSCILLATION FREQUENCY. O+JFQ  
37600 REM 37700 GOSUB 9700  
37800 GOSUB 30300  
37900 REM 38000 REM 2. AT THIS FREQUENCY, CALCULATE THE SIGN DF (AL-1)  
38100 REM 38200 SY = INT(SGN(AL-1))  
38300 IF SY=-1 THEN STOP  
38400 REM ESTABLISH INITIAL INCREMENTATION VALUE FOR FG  
38500 FG = FQ  
38600 DF = (FA-F1)/10#  
38700 GOSUB 30300  
38800 DE = (FG-F1)/10#  
38900 DF = 0  
39000 FG = F1
**STEADY-STATE ANALYSIS**

39100 REM The circuit model used in this analysis is similar to the one used
39200 REM in the small-signal analysis, but differs from it in two respects.
39300 REM First, it includes clamping and clipping effects described in the
39400 REM text. Second, the voltage source in the Thévenin equivalent of the
39500 REM amplifier is controlled by the input voltage in accordance with an
39600 REM input-output curve defined elsewhere in the program.
39700 REM The analysis applies a sinusoidal input signal of arbitrary
39800 REM amplitude, at the oscillation frequency, to the XTAL1 pin. Then
39900 REM calculates the resulting waveform from the voltage source. Using
40000 REM standard Fourier techniques, the fundamental frequency component of
40100 REM this waveform is extracted. This frequency component is then
40200 REM multiplied by the factor \(2L/(2L+RD)\), and the result is taken to be
40300 REM the signal appearing at the XTAL2 pin. This signal is then
40400 REM multiplied by the feedback ratio (\(\beta\)), and the result is taken to
40500 REM be the signal appearing at the XTAL1 pin. The algorithm is now
40600 REM repeated using this computed XTAL1 signal as the assumed input
40700 REM sinusoid. Every time the algorithm is repeated, new values appear at
40800 REM XTAL1 and XTAL2, but the values change less and less with each
40900 REM repetition. Eventually they stop changing. This is the steady-state.
41000 REM The algorithm is as follows:
41100 REM 1. Compute approximate oscillation frequency.
41200 REM 2. Call a circuit analysis at this frequency.
41300 REM 3. Find the quiescent levels at XTAL1 and XTAL2 (to establish the
41400 REM beginning DC level at XTAL1).
41500 REM 4. Assume an initial amplitude for the XTAL1 signal.
41600 REM 5. Correct the DC level at XTAL1 for clamping effects, if necessary.
41700 REM 6. Using the appropriate input-output curve, extract a DC level and
41800 REM the fundamental frequency component (multiplying the latter by
41900 REM \(2L/(2L+RD)\)).
42000 REM 7. Clip off the negative portion of this output signal, if the
42100 REM negative peak falls below zero.
42200 REM 8. If this signal, multiplied by (\(\beta\)), differs from the input
42300 REM amplitude by less than 1%. or if the algorithm has been repeated
42400 REM 10 times, exit the routine.
42500 REM 9. Otherwise, multiply the XTAL2 amplitude by (\(\beta\)) and feed it
42600 REM back to XTAL1, and go back to 5.
42700 REM 10 Compute approximate oscillation frequency.
46900 GOSUB 9700
47000 REM 2. CALL A CIRCUIT ANALYSIS AT THIS FREQUENCY
47200 GOSUB 20800
47300 PRINT PRINT "ASSUMED OSCILLATION FREQUENCY: "
47400 GOSUB 26600
47500 PRINT PRINT
47600 REM
47700 REM 3 FIND QUIESCENT POINT
47800 REM (At quiescence the voltages at XTAL1 and XTAL2 are equal. This
47900 REM voltage level is found by trial-and-error, based on the input-
48000 REM output curve, so that a person can change the input-output curve
48100 REM as desired without having to re-calculate the quiescent point.)
48200 VI = 0
48300 VB = 1
48400 K1 = 1
48500 VI = VI + VB
48600 GOSUB 13600
48700 IF ABS(VO-VI)<.001 THEN 49200
48800 IF K1+SGN(VO-VI)=0 THEN 48900 ELSE 48500
48900 K1 = SGN(VO-VI)
49000 VB = -VB/10
49100 GOTO 48500
49200 VB = VI
49300 PRINT "QUIESCENT POINT = ",VB
49400 REM
49500 REM 4. ASSUME AN INITIAL AMPLITUDE FOR THE XTAL1 SIGNAL.
49600 EI = .01
49700 NRX = 0
49800 REM
49900 REM 5. CORRECT FOR CLAMPING EFFECTS, IF NECESSARY
50000 REM (K1 and K2 are curve-fitting parameters for the ROM parts.)
50100 K1 = (2.5-VB)/(3-VB)
50200 K2 = (VB-1 25)/(3-VB)
50300 IF ICX=2 OR ICX=3 THEN IF EI<(VB+ 5) THEN ED = VB ELSE ED = EI - 5
50400 IF ICX=1 OR ICX=4 THEN IF EI<(VB+ 5) THEN ED = VB ELSE ED = K1*EI+K2
50500 NRX = NRX + 1
50600 REM
50700 REM 6. DERIVE XTAL2 AMPLITUDE
50800 VO = 0
50900 VC = 0
51000 VS = 0
51100 FOR NX = -25 TO +24
51200 VI = ED - EI*COS(PI*NX/25)
51300 GOSUB 13600
51400 VO = VO + VO
51500 VC = VC + VO*COS(PI*NX/25)
51600 VS = VS + VO*SIN(PI*NX/25)
51700 NEXT NX
51800 VO = VO/50
51900 VI = SQR(VC^2+VS^2)/25*FNZM(RL, XL)/FNZM((RL+RO), XL)
52000 REM
52100 REM 7. CLIP XTAL2 SIGNAL
52200 IF VO-VI<0 THEN VL = 0 ELSE VL = VO-VI
52300 PRINT PRINT "XTAL2 SWING = ",EO+EI TO ",EO-EI"
52400 PRINT "XTAL2 SWING = ",VL TO ",VL+VL"
52500 REM
52600 REM 8 TEST FOR TERMINATION
52700 IF ABS(EI-VI*B)<.001 OR NRX=10 THEN RETURN
52800 REM
52900 REM 9. FEED BACK TO XTAL1 AND REPEAT
53000 EI = VI*B
53100 GOTO 50300

230659-50
Intel’s 87C75PF Port Expander Reduces System Size and Design Time

TERRY KENDALL
MICROCONTROLLER PERIPHERALS
INTRODUCTION


What if a single component helped you achieve smaller board size, higher reliability, lower power, faster design time, simplified manufacturing, and lower cost? Intel's 87C75PF is the first in a family of microcontroller peripheral port expander products. This application note will show how the 87C75PF significantly reduces chip count and greatly simplifies system design. The 87C75PF data sheet has detailed device information.

Intel's early microcontrollers had obvious benefits over previous alternatives — a high degree of system integration. The most common microcomputer functions — CPU, ROM, RAM, I/O ports, timers/counters, address decoding, etc. — were combined onto a single chip. Upgrades and proliferations have grown significantly since those early days. Four-bit and 8-bit controllers are the most widely used, with 16-bit versions, spearheaded by Intel's 8096 family, beginning their exponential growth.

The most sought after microcontroller improvement is additional program memory. 8- and 16-bit controllers are optionally equipped with 4K or 8K bytes of ROM or EPROM. This is sufficient memory for about half of embedded applications.

The remaining applications use off-chip EPROM. One reason, of course, is to increase system memory; typically to 16K- or 32K-bytes. Another is to provide flexibility for code that changes frequently. In other applications, generic boards or multi-use modules can be manufactured and custom-programmed for special configurations. For example, a single robot control module can be manufactured. Identical robots can be configured to perform various factory tasks.

8- and 16-bit microcontrollers accommodate external-memory expansion. Controllers sacrifice two 8-bit I/O ports to supply address and data lines to peripheral components. Unfortunately, expanded-memory modes violate two embedded-control objectives: maximizing I/O capability and reducing chip count (or board size). Usually, systems that need more memory are also I/O intensive. Traditional memory-expansion/port-recovery schemes use multiple chips. Memory, address latches, port latches, transceivers, address decoders, and glue chips turn a single-chip uC system into a multiple-chip conglomerate.

THE MULTIPLEXED BUS

To achieve small board size, embedded control systems require minimum chip count and chips that occupy small footprints. Embedded controllers use multiplexed address/data buses to achieve both. An 8051 controller, for example, shares its lower eight address pins with its 8-bit data.

Every memory access requires two cycles — one for address, one for data (see Figure 1). The controller's first cycle places a 16-bit address on the bus. It holds the upper eight bits constant throughout the access. It presents the low-address byte just long enough for an external latch to capture it. The latch and controller's upper bus then supply the 16-bit address to external devices for the remainder of the memory access. The controller's data cycle transmits or receives data on its multiplexed lower address/data pins. The multiplexed bus minimizes the controller's pin count and the system's board traces.

![Figure 1. Every microcontroller memory access requires two cycles.](292048-1)
WHY A PORT EXPANDER?

Single-chip microcontroller solutions are quickly giving way to multiple-chip, high-end solutions. Embedded control applications often require more program memory than the microcontroller's on-chip memory. Sometimes, code flexibility is needed. The 87C75PF's 32K byte EPROM dwarfs any microcontroller's on-chip memory.

In the near future, microcontroller chip-sets — controller and peripheral — will make up most embedded control applications. The controller will contain features that must be coupled closely to its CPU. The peripheral chip will provide memory and I/O functions.

Controller and peripheral-chip costs will be more balanced. The chips will share complexity, which equates to cost. Two smaller, less complex chips will cost less than one huge controller chip, resulting in lower total system cost.

Typically, adding external functions to microcontrollers requires many chips and substantial board space. Address latches, memory, port recovery, and glue chips require far more space than a single-chip microcontroller. System reliability and performance are degraded. Design and manufacturing are more complicated.

Intel's high-performance 87C75PF Port Expander doesn't compromise designers' goals to create reliable, minimum chip systems. Its single chip, no-glue interface simplifies design and manufacturing while increasing performance and reliability — in the smallest possible board space.

A TYPICAL SYSTEM

Intel's 8051 microcontroller architecture is the most widely used. Many variations are available with enhanced I/O features and various amounts of memory. Intel's 80C31 is a non-ROM, CHMOS version of the 8051. It will help illustrate the 87C75PF's benefits over typical multiple-chip uC solutions.

Figure 2 shows a typical expanded microcontroller system. Whenever memory-mapped devices are connected to a microcontroller, two 8-bit ports lose their I/O functions to become address and data pins. Figure 2 shows port-reconstruction devices, a 256K-bit EPROM, and glue chips that make up an embedded control system. Nineteen chips are required!

![Figure 2. Many discrete chips provide EPROM and port expansion.](292048-2)
SYSTEM PERFORMANCE

Every system component influences performance. Performance encompasses speed, system noise, and power consumption. A typical expanded-mode controller application uses many chips to increase memory and recover lost I/O. Figure 3 shows an improved, but more expensive, alternative to the system in Figure 2. "Glue" chips between the controller and peripherals delay address signals. To optimize system speed, fast, expensive glue chips, memory, and peripheral devices are required.

Multiple-chip solutions consume significant power and inject noise into a system. A beefed-up, well regulated power supply will relieve symptoms, but adds significantly to cost, board size, and weight.

THE 87C75PF SOLUTION

Figure 4 shows the same system using the 87C75PF—a two chip solution!

The 87C75PF furnishes a no-glue interface to 8051-based systems and all other Intel-architecture embedded controllers. The Port Expander's flexible, user-progammable memory map and alterable control signals simplify 8051, 8096, and 80188 connections.

Examples in this application note show how the 87C75PF works with various microcontrollers. An 8051/87C75PF system that takes advantage of high-level compiled languages and an in-system programmable example will also be shown.

SYSTEM INTEGRATION

Intuitively we all recognize the benefit of system integration—chip-count is reduced.

Just as important are:
- small board size with few layers
- increased performance
- decreased design time
- optimized software development
- reduced inventory
- less incoming inspection
- increased system reliability
- simplified manufacturing.

Cost is a prime consideration. The itemized cost of discrete components is only one parameter. Until the benefits listed above are quantified, realistic system costs can't be determined. Hardware design and software development time are significant up-front expenses. Multiple-chip systems incur substantial inventory, incoming inspection, testing, manufacturing, board size, and rework costs.
Reliability also has significant value — to you and your customers. Customers demand products that work properly — forever. Reworked products waste time and money, increase the cost of every unit you ship, and ruin your company's reputation. The best way to increase reliability is to eliminate system components.

Simplified manufacturing saves time and money while increasing reliability. One factory-tested, integrated-function chip is much easier to place on a circuit board and is far more reliable than myriad discrete chips. Every solder joint is a possible failure point. A single chip reduces potential failure points from hundreds to a few.

**87C75PF ARCHITECTURE**

The 87C75PF Port Expander's features include:

- Two 8-bit I/O ports
- 32K × 8 EPROM
- Two 64K-byte memory planes
- Special Function Registers
- Device-configuration registers
- "No-glue" controller interface
- Low-power, Low-noise CHMOSII-E
- Quick-Pulse Programming™ Algorithm
- In-system programmability
- 40-Pin CERDIP, 44-Lead PLCC packages

**Two Ports**

The 87C75PF has two 8-bit bi-directional I/O ports. Port 1 has open-drain outputs and port 2 has quasi-bi-directional (resistor pull-up) outputs. Each port is individually addressable with separate port-latch and port-pin addresses. Typical of quasi-bi-directional ports, they are always in output mode but can be used as inputs by simply writing logic "1s" to their latches.

**Relocatable EPROM**

The EPROM has 262,144 bits organized as 32K 8-bit words. Its access time determines the device's speed rating. The 32K-byte EPROM occupies half of the program memory (or EPROM) plane. The EPROM block can be located in either the lower or upper half of the EPROM plane to accommodate various microcontroller architectures.

**Dual or Single Memory Planes**

8051-family microcontrollers have two external memory planes — program and data. 8096-, 80188-, and 68xx-family microcontrollers have only one program/data plane. The 87C75PF's user-configurable double- and single-plane modes work with any 8-bit microcontroller architecture.

**Relocatable SFRs**

The 87C75PF has five special function registers:

- Port 1 latch
- Port 2 latch
- Port 1 Pin
- Port 2 pin
- Plane select.

Port-latch registers allow the microcontroller to change port-pin output levels. The microcontroller can read the port latches to recall the last value written. A microcontroller can determine external pin levels by reading the port-pin locations.

During programming, the plane select register determines whether the EPROM array or the configuration registers are being programmed. More special function register details are described later in this application note.

**Device Reconfiguration**

Non-volatile (EPROM cell) device-configuration registers configure the 87C75PF for microcontroller compatibility. Programmable configuration registers can:

- relocate the EPROM array in the memory map
- relocate the SFRs in the memory map
- combine the EPROM and SFR planes
- change the reset pin's active polarity
- insert transistor pull-ups on port pins.
In its default configuration, the 87C75PF is compatible with the 8051's two-plane architecture. It is easily re-configured for single-plane 8096 architecture. Remapping the memory planes makes the device compatible with 80188 and 68xx architectures.

Various microcontrollers have different reset input levels. The 8051's reset is active-high while the 8096's is active-low. The 80188 has an active-low reset input and active-high synchronous reset output. The Port Expander's configurable reset polarity can work with active-high or active-low microcontrollers.

If the I/O ports are used only as outputs, a “push-pull” drive is desirable. Port 1 and/or port 2 can be configured to have active pull-up transistors rather than open-drain or quasi-bi-directional outputs.

"No-glue" Microcontroller Interface

The 87C75PF's internal address latches, address decoders, reconfigurable memory planes, and alterable control inputs allow no-glue interfacing to any Intel microcontroller. The 87C75PF makes expanded-mode, two-chip microcontroller systems a reality.

Quick-Pulse Programming

Intel's microcontroller, peripheral, and EPROM products employ the industry's fastest, most reliable Quick-Pulse Programming™ algorithm. Optimized Quick-Pulse Programming equipment can program the 87C75PF in four seconds.

In-circuit Programming

With its integrated features, the 87C75PF is easily programmed in-system. Built-in address latches, address decoders, and flexible control inputs enable the system's microcontroller to program the Port Expander. The section "80C51 In-system programming" describes this technique.

Packaging

For systems requiring periodic reprogramming, prototyping, or hermetic packages, the 87C75PF is available in a 40-pin ceramic DIP (CERDIP) package. PLCC packaging is available to further reduce board size and provide for surface mount and automated manufacturing.

Figure 5. 87C75PF Block Diagram.
**87C75PF FUNCTIONAL BLOCKS**

Figure 5 shows the 87C75PF's block diagram. The device has three main functional blocks, or memory planes: EPROM, special function registers, and configuration registers.

The block diagram shows device inputs on the left and outputs on the right. Sixteen address lines enter the device and their states are latched by ALE. The lower eight address pins are multiplexed with data. PSEN (Program-Store Enable) gates the device's EPROM data. RD gates SFR data. WR/PGM controls SFR data writes. CE is the master chip enable input. VPP (the programming voltage input) is multiplexed with RST (reset). VPP is required only during programming. Asserting RST sets port latches to “Is” during operating mode.

Port 1 is an 8-bit open-drain port with optional “CMOS” drive capability. Port 2 has 8 quasi-bi-directional pins, also with optional “CMOS” drive.

**DEVICE PINOUTS**

The 87C75PF is available in two package styles — 40-pin CERDIP and 44-lead PLCC. Both pinouts are similar to Intel's 27210 megabit EPROM. The device's pinouts are compatible with most programming equipment capable of programming 27210 EPROMs.

Figure 6 shows the CERDIP pinout. The left side has sequential address and data inputs. The ground pin (GND) separates lower and upper address lines for better noise immunity. Ports are logically placed on the device’s right side. Port 1, which is open-drain, is near VCC. SIP-pack resistor pull-ups added externally to port 1 have easy access to VCC.

Figure 7 shows the PLCC pinout. PLCC leads are in the same sequence as the CERDIP pinout. No-connect (NC) and don't-use (DU) leads are inserted at strategic locations. Future enhancements will use these leads for expanded features. DU leads should be left unconnected.
3 PLANE MEMORY MAP

The 87C75PF has three memory planes: EPROM, SFR, and configuration. Two planes, EPROM and SFR, are available during operating mode. The configuration plane is present under special programming conditions. Figure 8 shows the three memory planes, conditions when they are present, control signals that access them, and memory locations they occupy.

EPROM Plane

The 32K-byte EPROM fills the lower half (0000h-7FFFh default) of the 64K-byte EPROM plane. This conforms to 8051- and 8096-family microcontrollers that have reset and interrupt addresses in the bottom half of the memory map. The EPROM array can adapt to 80188- and 68xx-family microcontrollers by moving it to high memory (8000h-FFFFh). PSEN is the EPROM array's operating- and programming-mode read control. WR/PGM strobos data into the array only during programming mode.

SFR Plane

Special function registers are located in the SFR plane. They occupy low-addresses in a relocatable 2K-byte block (default addresses F800h-FFFFh). The 2K SFR block can be placed on any 2K-byte address boundary to match microcontroller architecture requirements. RD and WR/PGM control reads and writes from/to this plane.

Configuration Plane

The configuration plane contains non-volatile EPROM registers that determine the device's configuration. This plane is available only when high voltages are applied to special pins. PROM programming equipment can use this plane to identify the device, read its present configuration, and program new configurations. Memory-mapped registers can be programmed to:

Figure 8. The 87C75PF has three internal memory planes — SFR, EPROM, and Configuration.
• move the EPROM array
• move the SFR block
• combine the EPROM and SFR planes
• combine PSEN and RD
• change RST's polarity
• insert pull-up transistors on port output drivers.

Device reconfiguration will be covered further in the “Architecture compatibility” section.

Plane Select Register

The plane select register (PSR) occupies address F810h in the SFR plane (Figure 9). This register's value determines which plane, EPROM or configuration, is in programming mode. The following plane is programmed when Vpp is raised to its programming voltage if PSR contains:

- xxxxxx00 = programming prohibited
- xxxxxx01 = EPROM plane
- xxxxxx10 = configuration plane
- xxxxxx11 = programming prohibited.

Note that both PSR bits must toggle to change planes. Spurious programming noise is unlikely to alter both bits simultaneously. This safeguard prevents erroneous programming of the wrong plane.

I/O PORTS

The 87C75PF has two 8-bit, bi-directional I/O ports. Each port has two addresses in the SFR plane — port latch and port pin. The port latch register drives port pins; it's the port output register. Byte-wide data written to it is strobed by WR/PGM’s rising edge. This allows individual register bits to be changed without “glitching” unchanged bits. Port latches can be read to determine previously stored values. Redundant RAM locations that contain port values are not required. Asserting RST sets port latches to “1s”.

Each port has a pin register. This input register allows a microcontroller to monitor pin status. Although a port latch register may drive a port pin to “1”, an external switch can pull it to “0”. A software exclusive-OR of latch and pin values will discover the switch closure.

Figure 9 shows the 2K-byte SFR block (default location shown) containing port addresses. Locations F800h-F807h are reserved for port latch addresses; the 87C75PF uses only two of these addresses. Locations F808h-F80Fh are reserved for port pin addresses; again, the 87C75PF uses only two addresses. Each port latch and port pin register contains eight bits; each corresponding to a port pin. Locations F810h-F81Fh are reserved for SFR registers.
Port 1

Port 1’s default latch address is F800h; its pin address is F808h. Its default configuration is open drain. Other open-drain devices can be “wire-ORed” to port 1 pins.

Pull-up resistors can be added externally to provide $I_{OH}$ drive.

Port 1’s outputs can be reconfigured to supply CMOS drive. Programming the control level register’s P1C bit (CLR.6) inserts active pull-up transistors. This switches port 1 pins faster from $V_{OL}$ to $V_{OH}$ and simplifies interfaces to external CMOS devices. Figure 10 shows port 1’s block diagram.

Port 2

Port 2 is similar to port 1. Its latch address is F801h and its pin address is F809h. Its default configuration is quasi-bi-directional. This means that each pin has a weak pull-up resistor. External pull-up resistors can be added to increase the port’s $I_{OH}$ drive.

Port 2’s outputs can be reconfigured to supply CMOS drive. Programming the control level register’s P2C bit (CLR.5) inserts active pull-up transistors. Figure 11 shows port 2’s block diagram. Note the difference between port 2’s and port 1’s output stages. In addition to the weak pull-up resistor, the feedback network senses the pin’s $V_{OH}$ level and switches a stronger pull-up resistor into the circuit. A $V_{OL}$ level turns the resistor off. Another addition is the pulsed pull-up. When a port latch value changes from “0” to “1”, the CMOS transistor is pulsed to quickly supply current to the pin.

ARCHITECTURE COMPATIBILITY

Every microcontroller family has its own architecture. Each has unique boot-up, interrupt, and vectoring addresses. Some support dual external memory planes while others communicate with only one. External addressing capacity varies from 64K- to 1M-bytes.

The 8051’s control signals and software instructions manipulate 5 memory planes. Three planes are internal — on-chip ROM/EPROM, RAM/SFR, and bit-ad-
dressable registers. Two planes are external — program (EPROM) and data (RAM) memory. The instruction type drives internal and external read, write, and bus signals that select individual planes. An 8051 controller requires non-volatile boot-up memory, internal or external, at the bottom of its program memory plane. The 87C75PF’s two-plane external-memory architecture (see Figure 12) matches the 8051’s architecture. EPROM defaults to the EPROM plane’s low-memory and SFRs default to the SFR plane’s high-memory.

Figure 12. The 8051’s two-plane memory has reset and vector addresses in low program-memory.
8096-family controllers are typically used with a single 64K-byte external memory plane (Figure 13). Like the 8051, reset and vector addresses are in low memory. The 87C75PF has an optional single-plane configuration that complements 8096 architecture. The EPROM, located in low memory, is combined with the SFR plane.

Intel's 80188 microprocessor is used primarily in high-end embedded-control applications. Adding ports and memory makes the 80188 one of the most powerful microcontrollers available. The 87C75PF provides much of this hardware in a single package. The 80188 has a single memory plane. Unlike 8051 and 8096 controllers, its boot-up address is at the top of its 1M-byte address space (Figure 14). The 87C75PF can be configured for a no-glue 80188 interface.

The 87C75PF's flexibility simplifies hardware interfacing with many other microcontrollers. A 68xx controller, for example, has boot-up vectors at the top of its 64K-byte single-plane memory space. The Port Expander's memory map can be configured, much like that used by the 80188 (Figure 14), to accommodate 68xx controllers.

![Figure 13. The 8096 has a single memory plane.](image1)

![Figure 14. The 80188 boots up at the top of its 1M-byte address space.](image2)

**Default Configuration**

Ultraviolet light exposure will erase the 87C75PF's EPROM array and non-volatile configuration registers. The EPROM, SFRs, and other user-configurable options' default to:

- two memory planes — EPROM and SFR
- EPROM at 0000h-7FFFh
- SFR block at F800h-FFFFh
- reset (RST) active-high
- port 1 open drain
- port 2 quasi-bi-directional.
Changing the Reset Polarity

8051-family microcontrollers have active high reset inputs. 8096, 68xx, 80188, and special 8051-architecture controllers have active-low resets. The 80188 also has an active-high synchronous reset output.

The Port Expander’s alterable reset input (RST) can match any microcontroller. When erased, the 87C75PF’s RST is active-high. Programming the configuration plane’s control level register bit CLR.7 changes RST to active-low (see Figure 15).

Changing Port Output Drive

If port 1 and/or port 2 are used only as outputs, it may be preferable to have CMOS-type output levels. Programming CLR.6, P1C, and/or CLR.5, P2C (see Figure 15), inserts active pull-up transistors in port output buffers. These transistors supply higher current and faster switching than open drain or quasi-bi-directional outputs.

Moving the EPROM

The 87C75PF’s EPROM can be relocated to the upper half of its 64K-byte memory map. When erased, the EPROM is correctly positioned in low memory for 8051- and 8096-family controllers. Programming the configuration plane’s EPROM Location bit, ELR.7 (Figure 16), moves the EPROM to high memory for 80188 and 68xx compatibility.

Figure 15. The Control Level Register (CLR) determines the reset pin’s polarity and CMOS port drive.
Double- and Single-plane Configurations

The 87C75PF has two operating-mode memory-planes — EPROM and SFR. These planes share identical memory addresses. The EPROM plane is selected when PSEN is TTL-low. The SFR plane is selected when either RD or WR is TTL-low. 8051 microcontrollers use PSEN, RD, and WR to select two external memory planes. 8096 controllers have only RD and WR; some versions have an "INST" output that allows external circuitry to determine when instructions are being issued. Most other microcontrollers provide read and write signals that control only one memory plane.

Programming the 87C75PF’s overlap bit, OVLP (ELR.6), converts the device from dual-plane to single-plane (see Figure 16). When ELR.6 = “0”, PSEN and RD are internally combined. Both memory planes are active if either is TTL-low.

8051 applications that use code compiled from high-level languages find this especially useful. Some high-level languages can’t distinguish between data-plane and program-plane addresses. For example, look-up tables stored in the same EPROM as program instructions require PSEN to be asserted. However, a compiler interprets look-up table instructions as data fetches. It assigns code that asserts RD instead of PSEN. A typical hardware solution uses an AND gate to combine PSEN and RD. This forms one memory plane that is accessed by either signal. Programming the 87C75PF’s OVLP bit provides this “AND” function.

This bit also permits the SFRs to overlap the EPROM array. This allows multiple Port Expanders to be used in single-plane applications. For example, two Port Expanders can be used in an 8096 system (see Figure 22). Normally, two 87C75PFs’ 64K EPROM bytes consume the entire address space leaving no room for port addresses or external RAM. When ELR.6 = “0” and the device’s 2K-byte SFR block overlaps its EPROM array, 2K EPROM bytes are sacrificed to make room for the SFRs and external RAM. Under these conditions, the 87C75PF remains in a high impedance state during any access to the 2K-byte SFR-block except for the five valid SFR addresses (see Figure 9).

<table>
<thead>
<tr>
<th>ELR</th>
<th>OVLP</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELR.7 = EPROM LOCATION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 = High Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 = Low Memory (Default)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELR.6 = SFR/EPROM OVERLAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 = Overlap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 = No Overlap (Default)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>Overlap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>THESE BITS ARE DON'T CARE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 16. The EPROM Location Register determines the EPROM’s memory-map location
Moving the SFR Block

The 2K-byte SFR block's default location is F800h-FFFFh in the SFR plane. This location is fine for 8051 and 8096 applications. However, 80188 and 68xx-family controllers have boot-up and vector addresses in this address range; EPROM should be located here.

The SFR block can be moved to any 2K-byte device-address boundary. The SFR location register's (SFRLR) five bits determine the SFR-block's most-significant address bits. When erased, these bits are all “1s”, placing the SFRs at 11111xxx xxxxxxxxb or F800h-FFFFh. Programming the SFRLR to 01111xxx, for example, relocates the SFR-block to 7800h-7FFFh (just below the EPROM array when it's at the top of memory, 8000h-FFFFh). Programming SFRLR to 00000xxx moves the SFRs to the bottom of memory, 0000h-07FFh. Figure 17 shows the SFRLR and its bit definitions.

Programming the Configuration Plane

The 87C75PF data sheet describes detailed programming requirements. PROM programming equipment makes device reconfiguration easy. Down-loading EPROM code (from 0000h to 7FFFh) to the programmer is the same as for any 256K PROM device. The programmer allows editing of CLR, ELR, and SFRLR codes to reconfigure the device. Once programming commences, the EPROM array and the configuration registers are programmed automatically.

Figure 17. The SFRLR determines the 2K-byte SFR block's base address.
87C75PF APPLICATIONS

Now that you're familiar with how the Port Expander is organized and reconfigured, this section highlights some application examples. You'll see how the 87C75PF connects to 8051, 8096, 80188, and 68xx microcontrollers. Also shown are more sophisticated applications that use multiple Port Expanders and one that allows the microcontroller to program its own Port Expander. All of the applications illustrated show microcontroller/Port Expander interfaces, memory maps, and configuration register (CLR, ELR, SFRLR) values.

80C31 + 87C75PF

8051-family controllers usually operate in two-plane mode. To use external program memory (EPROM) exclusively, the controller's external access pin, EA, is tied to ground. Port 2 supplies upper addresses, A8–A15. Port 0 becomes the multiplexed lower-address/data bus, AD0–AD7. PSEN is the program memory read strobe. WR and RD (port pins P3.6 and P3.7) control external RAM and other read/write devices. RST is active-high on most 8051-family microcontrollers. Some special-purpose '51-based controllers have active-low resets.

Figure 18 shows a typical 80C31 + 87C75PF no-glue application. The 87C75PF's EPROM, SFR, and control-signal default-settings are already configured. Programming the large XX place holders shown in the CLR register enables CMOS port drive.

Figure 18. The 87C75PF's no-glue interface takes advantage of the 80C31's two-plane memory map.
80C31 + Two 87C75PFs

High-end applications, such as telecommunications, require sizable program memories and numerous I/O ports. Many of these applications use 8051-family microcontrollers. Two 87C75PF Port Expanders supply added I/O while furnishing EPROM — without using “glue” devices!

Figure 19 shows two Port Expanders in an 80C31 system. Port Expander 1’s EPROM is in its default low-memory location (0000h–7FFFh). Its SFR block is moved to F000h, out of Port Expander 2’s SFR range (F800h). Port Expander 2’s EPROM is moved to high-memory (8000h–FFFFh). Each device’s configuration register values are shown below the memory map. This configuration provides 16 additional I/O pins, 64K EPROM bytes, and leaves 60K for RAM and other memory-mapped devices.

Figure 19. Two 87C75PFs provide 16 I/O pins, 64K EPROM bytes, and room for 60K of RAM.
High-level Language 80C31 + 87C75PF

The 8051's two-plane flexibility challenges hardware and software engineers' creativity. Its two planes logically separate program and data planes to create 128K-bytes of memory in a 64K address space. However, many applications have look-up tables in non-volatile memory, usually in the same EPROM that contains program code. Unique assembly-language instructions drive hardware signals, PSEN, RD, and WR, to determine which plane is active.

Some compiled, high-level programming languages, however, have a hard time dealing with two-plane memories. They can't determine which 8051 instruction to use when look-up tables occupy the program plane. They usually assign an instruction that activates RD, rather than PSEN.

The typical solution forces the system to operate in single-plane mode by combining PSEN and RD with an AND gate. If either signal is TTL-low, the AND gate's output drives a common external-memory read signal. A compiler can now assign its typical "read from data memory" instruction.

The Port Expander has this "AND" function built in. Programming the configuration plane's Overlap bit, ELR.6, internally combines PSEN and RD; if either is at TTL-low EPROM or SFR data, depending on the address, is read. Figure 20 shows a typical high-level-language application.

Programming this bit also allows the SFR-block to overlap the EPROM in single-plane applications. If, and only if, these blocks overlap, 2K EPROM bytes are sacrificed to make room for the SFR block. The "8096 + two 87C75PFs" section illustrates this.

Figure 20. Programming ELR.6 combines PSEN and RD to form a single memory plane.
8096 + 87C75PF

8096-family 16-bit microcontrollers can also operate in 8-bit mode. These high performance controllers manage applications that are I/O intensive and, as a result, require large EPROM arrays. The 87C75PF expands the I/O while providing the EPROM.

The 8096 accesses a 64K-byte single-plane memory. Its memory map is similar to the 8051's. External EPROM is required at its low-memory boot-up location (2080h). The 87C75PF's EPROM and SFRs are appropriately located.

The 8096's reset input (RES) is active-low. Programming the Port Expander's reset level configuration bit, RSTL (CLR.7), makes RST's polarity active-low.

The 87C75PF is converted to single-plane mode by either tying PSEN and RD to the 8096's RD pin or by programming ELR.6, the overlap bit. If the latter option is chosen, the unused input, PSEN or RD, should be tied to VCC. Figure 21 shows a "no-glue" 8096 + 87C75PF application.

![Figure 21. The 87C75PF is also the no-glue Port Expander for 8096 systems.](image-url)
8096 + Two 87C75PFs

Single-plane 8096 applications can use two Port Expanders. Figure 22 shows this no-glue, three-chip system.

Port Expander 1 has its EPROM in default low-memory. Its SFR block is mapped over its EPROM; location 7800h is arbitrarily chosen. Programming 01111xxxxb into SFRLR moves the SFR block. Programming ELR.6 (to "0") overlaps the EPROM and SFR planes; one plane is formed. This bit also tells the Port Expander that its SFRs are intentionally mapped over its EPROM. The device sacrifices 2K EPROM bytes to make room for the SFR block. Any access to this 2K-byte block, except valid port and PSR addresses, places the external data bus in a high impedance state. External RAM can occupy the 2K-byte space.

Port Expander 2 is also reconfigured. Its EPROM is moved to high-memory by programming ELR.7. Its SFR block must overlap its EPROM array; 8000h is arbitrarily chosen. Port Expander 2's overlap bit, ELR.6, is programmed to form a single plane and to tell the device that its SFRs are intentionally mapped over its EPROM, like Port Expander 1. This configuration supplies four additional 8-bit ports, 60K EPROM bytes, and still leaves 4K bytes free for RAM.

Figure 22: Two 87C75PFs add 16 I/O pins, 60K EPROM bytes, and leave room for 4K of RAM.
**80C188 + 87C75PF**

The 80C188 found its niche in high-end embedded control applications. This CPU, when combined with RAM and the Port Expander, becomes a powerful embedded controller. Its 1M-byte address range accommodates several Port Expanders and large amounts of RAM. Although the 80C188 has two planes, memory and I/O, the Port Expander works best in the memory plane. Figure 23 shows a simple 80C188 + 87C75PF system.

The 80C188 boots up at address FFFFOh. The 87C75PF’s EPROM array is moved to its high memory (8000h–FFFFh) by programming ELR.7. The SFR block must be moved to lower memory outside of EPROM-block addresses, (F7800h is shown). Pro-ramming the overlap bit, ELR.6, or tying PSEN and RD to the 80C188’s RD combines the EPROM and SFR planes. The processor’s UCS, connected to the 87C75PF’s CE, selects the Port Expander in the upper address range. The 80C188’s reset input, RES, is active low. Programming the 87C75PF’s RSTL bit, CLR.7, converts RST to active-low. the 80C188 also has an active-high synchronous reset output. This output can be connected to the 87C75PF’s RST without reconfiguring RST’s polarity.

80C188 systems usually have larger RAM arrays than typical microcontroller applications. Figure 23 shows the simple RAM interface. The RAM does not contain its own address latches, so an 8-bit latch must be used to capture addresses A0–A7.

---

Figure 23. The Port Expander and SRAM make the 80C188 a powerful embedded controller.
68xx + 87C75PF

The microcomputer industry's peripheral- and memory-interface standard dictates chip-enable, output-enable, and write-enable polarities. All are active-low. The 87C75PF conforms to this industry standard.

Like Intel controllers, 68xx-family microcontrollers use multiplexed address/data pins. However, they differ in two significant ways. First, 68xx controllers have high-memory reset- and interrupt-vector addresses. Address A15 is logic-high during vector accesses. Second, read and write controls are functions of R/W and E (clock output). Combinational logic must convert R/W and E to industry-standard RD and WR signals.

The 87C75PF's memory map can be reconfigured and its two memory planes combined to simplify 68xx interfaces. Its RST polarity can match a 68xx's active-low reset. All that's required to complete the interface is to condition R/W and E to RD and WR. Figure 24 shows a 68xx + 87C75PF system and its memory map.

Figure 24. One NAND-gate package interfaces the 87C75PF to 68xx controllers.
AP-318

PROGRAMMING

EPROM and Configuration Registers

PROM programming equipment makes the 87C75PF as easy to program as EPROM-version microcontrollers and standard EPROMs. Optimized programming equipment that utilizes the Quick-Pulse Programming™ algorithm can program the 87C75PF in less than four seconds.

Data I/O's model 29B (version V06), with Unipak-2B module (version 16, family/pin code = 112/107) and 87C75PF cartridge, supports the 87C75PF. It has a straightforward programming procedure. Assembled code is transferred to programmer RAM addresses 0000h-7FFFh. Configuration registers (CLR = 7FFDh, ELR = 7FFEh, and SFRLR = 7FFFh) are loaded into programmer RAM addresses 8000h, 8001h, and 8002h. Configuration register contents can be entered manually using the programmer's edit command.

With EPROM and configuration register contents loaded, the programmer automatically programs the EPROM array and non-volatile registers. The programmer can also read a programmed master device's EPROM array and configuration registers and program duplicates without further editing. Contact Data I/O or your programmer vendor for further details.

80C51 In-system Programming

Factory programmed and field updated applications use in-system and board-programming techniques. Board programming equipment supplies voltages, addresses, data, and pertinent control signals to the board's edge-card connector.

In-system programming, on the other hand, allows a resident ROM- or EPROM-type microcontroller to program the system's off-chip non-volatile memory. A small amount of the microcontroller's ROM or EPROM contains code that controls its serial communications channel and knows how to program external EPROM.

Figure 25. A simple circuit allows the microcontroller to program the 87C75PF in-system.
Multiple-application modules can be customized using in-system programming. For example, a generic control module can be built, installed in a variety of end products, and customized for different tasks at the end of the production sequence.

Figure 25 shows a simple 80C51-based in-system-programmable module. The microcontroller’s on-chip ROM or EPROM contains the communication and programming algorithms. Port pins P3.0 and P3.1 provide the serial communication link. P3.2 (EACONT) controls the EA pin. When high (which occurs at reset or when "1" is written to it), internal program memory supplies code. When low, external EPROM supplies code. P3.4 (ALECONT) controls the ALE latching signal during programming. P3.5 (PGMON) controls programming and operating-mode VPP and VCC voltages. P3.6, which is the WR signal during normal operation, serves as the program pulse strobe, PGM, during programming. RD, P3.7, or PSEN can be used to verify programmed data whenever VPP is at its programming voltage.

PGMON also controls the ALE circuit. When PGMON is high, the microcontroller’s ALE value passes to the 87C75PF’s ALE pin. When PGMON is low, the microcontroller’s ALECONT controls ALE.

Figure 26 shows the program and latch control circuit. 5 volt and 12 volt supplies are connected to this circuit at all times. Inverter 74'06a allows 12 volts to pass into the DC/DC converter and the LM317 voltage regulators only when system power is on. PGMON is high after reset or when P3.5 contains a "1." PGMON controls inverter 74'06b which turns VPP on or off. Inverter 74'06c keeps VCC at 5 volts until programming commences. When PGMON goes low, these inverters turn off allowing VPP and VCC voltages to attain their programming levels. The variable resistors adjust VPP and VCC read- and program-voltages. VCC read voltage is 5.0V and its program voltage is 6.25V. VPP read voltage is off, so it doesn’t interfere with the 87C75PF’s reset, and its program voltage is 12.75V.

Figure 26. A microcontroller can use this circuit to control programming voltages and ALE.
The microcontroller's AD0−7 and A8−15 (ports 0 and 2) connect to the 87C75PF's AD0−7 and A8−15 pins. The controller's program-memory read signal, PSEN, controls the 87C75PF's output-enable, PSEN.

During programming, the controller brings EACONT high and PGMON low. This allows it to operate from internal code, enables programming voltages on the 87C75PF's VPP and VCC pins, and switches ALE control from the controller's ALE to its ALECONT. It then inputs data over its serial channel. With ALECONT high, an address is placed on ports 0 and 2. When ALECONT is brought low, the 87C75PF internally latches the address. Data read from the serial port is written to port 0. Port 0 must have pull-up resistors when used in its I/O port mode. The Port Expander now has both address and data information. The controller needs only to bring its WR pin low to program data into the addressed location.

The in-system programming sequence is summarized below.

1) Set EACONT = "1". Code is now supplied from the controller's internal program memory.
2) Assert PGMON. This switches VPP and VCC to their program voltages and allows the controller to manually control ALE via ALECONT. ALECONT and WR are high.
3) Down-load address and data information via Port 3's serial channel. Ports 0 and 2 serve as I/O ports, so place the 16-bit address on them. Bring ALECONT low to latch the address into the 87C75PF.
4) Write data information to port 0.
5) Bring WR low to program data into the 87C75PF. See the 87C75PF data sheet for the programming algorithm and timing requirements.

6) Verify the programmed data. When the 87C75PF's VPP is at 12.75V, its PSEN and RD pins are internally combined. The "MOVC A,@A+DPTR" instruction uses the PSEN pin to read EPROM data (or the "MOVX A,@DPTR" instruction uses the RD pin).

7) Repeat this sequence until all EPROM data is programmed and verified.
8) When programming is complete, de-assert PGMON and ALECONT. When EACONT = "0", code execution commences from the 87C75PF. Code duplication at identical internal and external memory locations allows uninterrupted paging between these two memory spaces.

When 6.25V is applied to the 87C75PF's VCC during programming, its port outputs, when "1", will be close to 6.25V. Careful system design should ensure that microcontroller and other device inputs can handle this elevated voltage. Writing "0s" to all port pins before VCC receives 6.25V will prevent damage to external devices.

**SUMMARY**

System demands push single-chip microcontroller designs to their limits. Complex applications are I/O intensive and use lots of EPROM. Traditional solutions use discrete chips – EPROM, address latches, address decoders, I/O port chips, and "glue" logic – to get more memory and expand, or recover, I/O.

Intel's 87C75PF Port Expander puts port functions, EPROM, and "glue" into a single package. Chip count and board size are dramatically reduced. System performance is optimized. Reliability is assured. Design time is shortened. Manufacturing is simplified. Device inventory is reduced.

Miniaturized system designs that weren't possible before, can now come to life, thanks to the 87C75PF.
Latched EPROMs Simplify Microcontroller Designs

TERRY KENDALL
MICROCONTROLLER PERIPHERALS
INTRODUCTION


This application note will show how Intel's latched EPROMs minimize board space and cost, simplify design and manufacturing, and increase performance and reliability in microcontroller systems.

A few years ago an embedded control system consisted of many discrete components. A general purpose microprocessor was combined with memories, timers, counters, I/O expanders, address decoders, latches, and assorted glue chips to make a basic control system. Then came the microcontroller. These functions, and many more, are now combined into a single chip.

Today, engineers are stretching the limits of microcontroller features. Controller manufacturers are stuffing as many functions and as much memory as die and package can accommodate. Microcontrollers typically have EPROM (or ROM) densities of 4K or 8K bytes; some advanced controllers even have 16K. Still, more is required.

Microcontroller applications are now moving back to multiple chip solutions. 32K-byte EPROMs are common in many medium and high-end systems. It is not practical to put this much memory on the microcontroller die; chip price becomes prohibitive. Most controllers have an expansion mode that allows external memory to be added.

Higher density is not the only reason to go “off-chip” for memory. Many systems are designed to be generic modules. For example, one engine control module can be designed for an entire line of car models. During a final manufacturing step the module can be custom programmed for any particular vehicle. ROM-version controllers don’t lend themselves to this application. EPROM memory allows any application to be customized — at any step in the manufacturing process.

But, using off-chip memory shouldn’t detract from the designer’s goal to achieve a minimum-chip system. Latched EPROMs provide microcontroller memory expansion without adding “glue” chips.

THE MULTIPLEXED BUS

To achieve small board space, embedded control systems require not only minimum chip count but chips that occupy small footprints. Embedded controllers achieve this by using multiplexed address/data buses. An 8051 controller, for example, shares its lower eight address pins with its 8-bit data.

Every memory access requires two cycles — one for address, one for data (see Figure 1). The controller

![Figure 1. Every microcontroller memory access requires two cycles.](image-url)
places a 16-bit address on the bus during the first cycle. It holds the upper eight bits constant throughout the access. It presents the lower address byte just long enough for an external latch to capture it. The latch and controller's upper bus supply the address to external devices for the remainder of the memory access. The controller uses its multiplexed lower address/data pins to transmit or receive data during the data cycle. As well as minimizing the controller's pin count, the multiplexed bus requires fewer board traces.

Before latched EPROMs, adding external memory to microcontrollers consumed excess board space. Address latches plus EPROM required more space than the controller itself. The address latch consumes significant board space and system power, degrades system reliability and EPROM performance, and complicates design and manufacturing.

Intel's high-performance latched EPROMs don't compromise designers' goals to produce minimum chip systems. The address latching function is built into the EPROM chip. The no-glue controller-EPROM interface simplifies design and manufacturing while increasing performance and reliability — in the smallest possible board space.

**MICROCONTROLLER MEMORY INTERFACE**

A typical microcontroller/memory interface is shown in Figure 2. Eight-bit controllers require at least one 8-bit address latch; Sixteen-bit controllers require two. In an 8-bit system, the controller's A8–15 address pins are connected directly to the EPROM's upper address pins. Address/data pins AD0–7 are connected to the EPROM's DO–7 data pins and to the address latch's inputs. The latch's outputs drive the EPROM's A0–7 address inputs. The controller's address-latch-enable, ALE, controls the latch. Figure 2a shows this memory interface.

Figure 2b shows a simplified system that uses a latched EPROM. All of the controller's bus signals connect directly to the latched EPROM. It's easy to see that design time (and manufacturing) are simplified. Performance is improved because latch propagation delay is non-existent. System reliability is assured — one factory-tested, integrated memory device is inherently more reliable than several discrete components.
Discrete latch chips, like the 74HCT573, have large output drivers. This allows them to drive many devices on a system's address bus. Unfortunately, large output drivers consume considerable power. Typical microcontroller applications are minimum-chip systems. Discrete address latches unnecessarily waste system power with their large drive capability. Intel's latched EPROMs use very little power because their built-in latches drive only internal address lines. Integrated address latches allow "no-glue" interfacing to 8-bit and 16-bit microcontrollers.

**SYSTEM INTEGRITY**

An address latch and associated board traces require about .75 inches². This doesn't sound like much, but compared to the EPROM's 1.2 in² and the controller's 1.5 in² it amounts to 22% of a system's board space.

Not only does a latched EPROM produce a more "elegant" design, but system reliability is improved. Every board component is subject to failure. A discrete latch requires twenty additional PC-board solder joints — each a potential failure point. Failures decrease as part count (elimination of latches) goes down.

Every board trace and component node is a source (or receptor) of system noise. Noise can degrade performance and compromise data integrity. EPROM performance requires rock-steady address inputs. When EPROM output buffers turn on, address input buffers are affected. A small ground reference fluctuation changes the threshold voltage of input buffer transistors. This can effectively change the EPROM's address in mid-access; data integrity is compromised.

Latched EPROMs are virtually immune to ground-reference shifts. Current surge caused by switching output buffers may affect the EPROM's address inputs, but the internally latched address remains steady; noise isn't transferred to address decoders. Access time and data integrity are optimized.

---

**Figure 3.** Propagation delays can be significant when standard EPROMs are used in uC systems. Latched EPROMs eliminate these delays.

---

9-83
SYSTEM PERFORMANCE

Latched EPROMs improve system performance. Discrete latches have inherent propagation delays. In a pure CMOS system, this delay is significant; a 74HCT373 latch delay is 45ns at automotive and military temperatures. A 16MHz 80C31 microcontroller, for example, provides 207ns for EPROM access time. A 45ns latch delay degrades this access time to 162ns. An EPROM rated at 160ns or faster must be used. Figure 3 shows the timing delays inherent in discrete component solutions.

If a latched EPROM is used, no external latch delay occurs. A 200ns latched EPROM can be used. Access time parameters include internal latch propagation delays. Slower, less expensive latched EPROMs do the same job as fast EPROMs and discrete latches.

ARCHITECTURE COMPATIBILITY

Intel's latched EPROMs have separate address and data pins. All address inputs contain latches. This simplifies 16-bit microcontroller interfacing. Pin layout is virtually identical to standard EPROMs. Upgrade-compatible circuit board designs are simplified. In 8-bit multiplexed address/data systems, EPROM pins A0-7 are connected directly to corresponding D0-7 pins. In 16-bit multiplexed systems, low-byte EPROM data pins D0-7 are connected to address lines A0-7 while high-byte EPROM data pins D0-7 are connected to address lines A8-15. See Figures 7 and 9 for typical 8-bit and 16-bit system examples.

THE LATCHED EPROM FAMILY

Intel's growing family of latched EPROMs includes the 87C64, 87C257, and 68C257. Ceramic DIP and PLCC package pinouts are shown in Figures 4 and 5. This application note shows how latched EPROMs simplify microcontroller system designs.

![Figure 4. 28-pin ceramic DIP latched-EPROM pinouts](image-url)
Figure 5. 32-Lead PLCC latched-EPROM pinouts.
87C64

The 87C64 is a 64K-bit EPROM organized as 8192 8-bit words. Integrated address latches make the 87C64 EPROM unique. This device is functionally identical to two 74HCT573 latches and a 27C64 EPROM (see Figure 6). However, with latches included, the 87C64 conserves:

- chip count
- system performance
- board space
- power consumption
- system cost
- inventory
- design time
- incoming inspection

In discrete component solutions, separate latches are used with a 27C64 EPROM. Even when the EPROM is in standby mode, the latches are always active, consuming full power. The 87C64 achieves low standby power in a novel way. It has a combined ALE/CE signal. When this signal is TTL-high, both the EPROM and the internal latches are placed in low-power standby mode. When ALE/CE is TTL-low, the latches activate, address information is latched, address decoding begins, and the EPROM is ready to present data at its outputs.

The 87C64 easily connects to an 80C31 microcontroller. EPROM data pins are connected to its A0–7 ad-

Figure 6. 87C64 Functional Diagram.
dress pins, which in turn connect to the controller's AD0-7 pins. ALE/CE must be generated by the processor's ALE signal, as shown in Figure 7. When ALE is high, a new address can flow into the device's latch. The address is latched when ALE goes low. EPROM data is present on AD0-7 when OE goes low.

**Using Multiple 87C64s**

If multiple chips are used in a low power system, address lines and the ALE signal are combined via an address decoder as shown in Figure 8. Connecting the ALE signal to the address decoder is important because the 87C64's ALE/CE input must toggle high-to-low each time the address changes.

The EPROM contains system operating code. The microcontroller typically accesses sequential addresses as it executes instructions. Upper address lines are used to decode memory blocks, but they usually don't change when sequential addresses are generated. This means that the outputs of an address decoder connected to these lines will not toggle as sequential addresses change. The address decoder shown in Figure 8 is gated by ALE to provide the latching signal at the 87C64's ALE/CE input.

---

**Figure 7. The 87C64 easily connects to the 80C31.**
87C64s in 16-bit Systems

The 87C64 is an ideal memory for word-wide systems. Two devices provide low-byte and high-byte data. Figure 9 shows an 8096 system that uses two 87C64s.

Microcontroller address/data lines AD13 are connected to address inputs A0-12 on both EPROMs. Address/data line AD0 is normally used to select low-byte data in read/write memories. This line need not be connected to read-only (EPROM) memories. In order to operate from external EPROM mapped at low-memory, the 8096's EA pin must be tied to ground.

The low-byte EPROM's DO-7 outputs are connected to the controller's AD0-7 lines. The high-byte EPROM's DO-7 outputs are connected to lines AD8-15. The controller's RD and ALE lines are connected directly to both EPROMs' OE and ALE/CE inputs.

In-System Programming

EPROMs are not just read-only memories, they're user-programmable. That's the reason EPROMs are the preferred non-volatile memory. EPROMs are usually programmed in PROM programming equipment. In-system programming, however, is becoming popular in applications that require factory programming or field updates.

In-system programming allows the resident microcontroller to program the system's EPROM. A small amount of the microcontroller's ROM or EPROM can contain code that knows how to down-load data over its serial channel and program an 87C64.

In-system programming allows a multi-use module to be customized for different applications. For example, a generic robot-control module can be built, installed in several locations, and customized for any particular job on an assembly line.
Figure 9. Two 87C64s provide a no-glue EPROM solution for word-wide systems.

Figure 10. A simple in-system programmable module.
Figure 10 shows a simple 80C51-based in-system programmable module. The microcontroller's on-board ROM or EPROM memory contains the communication and programming algorithms. Port pins P3.0 and P3.1 provide the serial communication link. P3.2 (EACONT) controls the EA pin. When high (which occurs at reset and when "1" is written to P3.2), code operates from internal memory. When low, external EPROM supplies code. P3.3 (ALECONT) controls the ALE latching signal during programming. P3.4 (PGMCONT) controls the 87C64's PGM (program pulse) pin. P3.5 (VPPON) controls the VPP and VCC programming and operating voltages.

Figure 11 shows the program and latch control circuit. The 5 volt and 12 volt supplies are connected to this circuit at all times. Inverter 74'06a allows 12 volts to pass into the DC/DC converter and the LM317 voltage regulators only when 5 volts is applied. VPPON is high at reset or when P3.5 contains a "1." Inverters 74'06b and 74'06c keep VPP and VCC at 5 volts until programming is initiated. When VPPON goes low, these inverters turn off allowing VPP and VCC voltages to go to their programming levels. VPP and VCC read- and program-voltages are adjusted by the variable resistors shown. VCC read voltage should be 5.0V and its program voltage should be 6.25V. VPP read voltage should be 5.0V and its program voltage should be 12.75V.

VPPON also controls the ALE circuit. When VPPON is high, the microcontroller's ALE value passes through to the 87C64's ALE/CE pin. When VPPON is low, ALE/CE can be controlled by the microcontroller's ALECONT signal during programming.

The microcontroller's A0-12 outputs are connected to the 87C64's A0-12 pins. The EPROM's D0-7 are connected to the controller's AD0-7 pins. The controller's program-memory read signal, PSEN, controls the 87C64's output-enable, OE.

![Figure 11. Program- and latch-control circuit for in-system programming.](image-url)
During programming, the controller brings \texttt{EACONT} high and \texttt{VPPON} low. This allows it to operate from internal code, enables programming voltages on the 87C64's \texttt{Vpp} and \texttt{VCC} pins, and switches \texttt{ALE/CE} control from the controller's \texttt{ALE} to its \texttt{ALECONT}. It then inputs data over its serial channel. With \texttt{ALECONT} high, an address is placed on ports 0 and 2. When \texttt{ALECONT} is brought low, the 87C64 internally latches the address. Data read from the serial port is then written to port 0. The 87C64 now has both address and data information. The controller needs only to bring \texttt{PGMCONT} low to program data into the addressed location.

The in-system programming sequence is summarized below.

1) Assert \texttt{EACONT}. Code is now supplied from the controller's internal program memory.

2) Assert \texttt{VPPON}. This switches \texttt{Vpp} and \texttt{VCC} to their program voltages and allows the controller to manually control \texttt{ALE} via \texttt{ALECONT}. \texttt{PGMCONT} and \texttt{ALECONT} are high.

3) Input address and data information from Port 3's serial channel. Ports 0 and 2 serve as I/O ports. Place the address on ports 0 and 2. Bring \texttt{ALECONT} low to latch the address into the 87C64.

4) Write data information to port 0.

5) Bring \texttt{PGMCONT} low to program data into the 87C64. See the 87C64 data sheet for the proper programming algorithm and timing requirements.

6) Verify the programmed data. Use the "\texttt{MOVC A0, @A + DPTR}" instruction to read EPROM data. The configuration shown in Figure 10 allows the 87C64 to be read at any 8K-byte boundary. This allows the controller to operate using its internal low-memory code and still verify external EPROM mapped at the same locations.

7) Repeat this sequence until all EPROM data bytes are programmed and verified.

8) When programming is complete, \texttt{VPPON}, \texttt{PGMCONT}, and \texttt{ALECONT} should be de-asserted. When \texttt{EACONT} = “0”, code execution will commence from the 87C64. Duplication of code at identical internal and external memory locations will allow uninterrupted paging between these two memory spaces (see application note AP-284 "Using Page-Addressed EPROMs" for further details).

Care should be taken during system design to ensure that microcontroller and other device inputs can handle elevated voltages supplied by the EPROM during programming. When 6.25V is applied to the 87C64's \texttt{VCC}, its outputs, when “1”, will be close to 6.25V.

\textbf{87C257}

The 87C257 is a 256K-bit EPROM organized as 32768 8-bit words. It also contains the equivalent of two 74HCT573 address latches. All address inputs are latched. Figure 12 shows the 87C257's block diagram. To serve high-performance 8-bit microcontrollers, the 87C257 has separate \texttt{ALE} and \texttt{CE} inputs. The 87C257 is pin compatible with the 27C256 (see Figure 4).

The \texttt{ALE/VPP} input serves as the latch enable during read mode and as the high voltage input during programming. When \texttt{ALE} is high, address information on pins \texttt{A0-14} flows through the latches to the input decoders. If \texttt{CE} is asserted (\texttt{CE} = \texttt{VIL}), the EPROM is in its active mode which allows address decoding to begin immediately. If \texttt{CE} is high, the 87C257 is in stand-by mode, but addresses can still be latched. The address latches retain present address-pin values when \texttt{ALE} goes low (\texttt{ALE} = \texttt{VIL}).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{87C257_functional_diagram.png}
\caption{87C257 functional diagram.}
\end{figure}
The 87C257 interfaces to 8051-family microcontrollers without “glue” chips. Figure 13 shows a simple 80C31/87C257 system. Note that all 8051-family controllers have similar interfaces. The 80C31’s port 0 serves as the multiplexed low-order address/data bus when used in expanded memory mode; port 2 is the high-address bus.

Port 0 pins connect directly to the 87C257’s A₀₋₇ and D₀₋₇ pins. Port 2 pins are connected to the 87C257’s A₈₋₁₄ and CE pins. Since the 87C257 fills the lower half of the 80C31’s program-memory map (0000h – 7FFFh), address line A₁₅ (P2.7) can be connected to the 87C257’s CE input. The EPROM is selected whenever A₁₅ is low.

The controller’s PSEN output is the program (or instruction) memory read-strobe. This pin is connected to the 87C257’s output enable pin, OE.

The 80C31’s ALE controls an external address latch. When ALE is high, the controller’s port 0 and port 2 pins present address information. When low, addresses A₀₋₇ are externally latched. The external latch then supplies the low-address to external memory devices. Since the 87C257 has its own latch, the 80C31’s ALE is connected to the 87C257’s ALE/VPP (the 87C257’s VPP function is internally disabled in read mode).

The 80C31’s EA (External Access) pin must be connected to ground when accessing external program memory between addresses 0000h and 0FFFh (the upper address boundary may vary depending on the 8051 version used).

Figure 13. A “no-glue” 80C31/87C257 system.
Two 87C257s + 80C31

8051-family controllers are unique in that two 64K-byte memory spaces can be addressed. These controllers have separate PSEN and RD signals that access program memory (ROM or EPROM) and data memory (RAM and peripheral devices). All system devices see the controller’s 16-bit address. Depending on the instruction type, either PSEN or RD is asserted. Although two devices can be memory mapped at identical locations, PSEN and RD determine which will present data.

Figure 14 shows two 87C257s in an 80C31 system. Each 87C257 connects to the 80C31 just as it did in the 87C257 + 80C31 example shown in Figure 13. The only difference is the inverter between A15 and the second 87C257’s CE. This inverter allows the second 87C257 to be selected when A15 is high — addresses 8000h – FFFFh.

![Diagram of 87C257 and 80C31 system](image-url)
Two 87C257s completely fill the 80C31's program memory space. A system that requires 64K-bytes of EPROM is probably performing complex I/O tasks. These tasks usually require more RAM than the microcontroller contains. Also, since the 80C31 loses two 8-bit I/O ports when accessing external memory, port reconstruction is desirable.

The 8155 shown in Figure 14 recovers the lost ports (plus 6 additional port pins) and supplies 256 bytes of RAM. In addition, it provides a 14-bit counter/timer. Connected as shown, the 8155's RAM is mapped at locations 0000h – 00FFh. Ports and timer addresses are mapped at 0100h – 01FFh. Since the 8155 is not fully decoded, shadow addresses occur at 512-byte boundaries.

The system shown in Figure 14 consists of a high-performance microcontroller, 64K-bytes of EPROM, 256 bytes of RAM (in addition to the uC's RAM), 36 I/O port pins, and an additional timer/counter. The only "glue" device in this system is the inverter, which can be made from one transistor and a resistor.

### 87C257 + 8096

Intel's 8096-family microcontrollers contain six 8-bit I/O ports, a powerful CPU, and many other high-performance features. 8096BH, 8098, and 80C196 versions also have 8-bit external bus modes that simplify interfaces to 8-bit memories and peripherals. When used in expanded mode, ports 3 and 4 supply the multiplexed address/data bus.

Figure 15 shows a no-glue 8096/87C257 interface. The 8096's EA (External Access) and Buswidth pins are tied to ground. This tells the controller that program-memory accesses are from external EPROM and that the external data bus is 8 bits wide.

Port 3 supplies multiplexed address/data information. Its pins are connected to the 87C257's A0-7 and D0-7 pins. Port 4 supplies addresses A8-15. Its pins are connected to A8-14 and CE. The EPROM is selected whenever A15 is low (addresses 0000h – 7FFFh), which encompasses the 8096's boot-up and vector locations. RD and ALE are connected to the 87C257's OE and ALE/VPP pins.

![Figure 15. An 87C257 enhances the powerful 8096.](292045-16)
The microcomputer industry has a standard for memory and peripheral interfaces which dictates chip-enable and output-enable polarities. Customers using non-standard-bus controllers asked Intel to provide a "no-glue" EPROM for their applications — the 68C257.

Like Intel controllers, 68xx-family uCs use multiplexed address/data pins. However, they differ in two significant ways. First, 68xx controllers use high-memory addresses for reset- and interrupt-vectors. Since A15 is high during vector accesses, it can't be connected directly to a standard EPROM's CE — an inverter is required. Second, read and write controls are functions of R/W and E (clock output). Fortunately, EPROMs don't require combinational logic to decode R/W and E. The active-high E output can simply be inverted before connecting it to an EPROM's OE input.

The 32K-byte 68C257 EPROM's inputs contain latches, just like the 87C257. The 68C257 also internally inverts CE and OE. Figure 16 shows the 68C257's block diagram. Figure 17 shows a no-glue 68C257/68xx interface.
**SUMMARY**

The best system design is small in size, easy to manufacture, highly reliable, and cost effective. Components that simplify the design process add even more value to the system.

Intel's latched EPROMs reduce chip count and board space, enhance performance, increase reliability, minimize design time, and simplify microcontroller systems. Latched EPROMs are available in popular 64K- and 256K-bit densities, and a version is available that will provide a "no-glue" interface to virtually any microcontroller architecture.

**Figure 17. The 68C257 is the "no-glue" EPROM for alternate-architecture microcontrollers.**
DOMESTIC SALES OFFICES

ALABAMA
Intel Corp. 5015 Bradford Dr., #2
Huntsville 35805
Tel: (205) 330-9210 FAX: (205) 837-2640

ARIZONA
Intel Corp. 11225 N. 28th Dr.
Suite D-214
Phoenix 85029
Tel: (602) 869-4980 FAX: (602) 869-4950

ALABAMA
Intel Corp. 1161 N. El Dorado Place
Suite A
Telfair 36715
Tel: (205) 292-8815 FAX: (905) 296-6234

CALIFORNIA
Intel Corp. 21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8800 FAX: (818) 240-1144

Intel Corp. 225E Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 640-6040 FAX: (213) 640-7133

Intel Corp. 1510 Arden Way
Suite 10
Sacramento 95815
Tel: (916) 920-8098 FAX: (916) 920-8253

Intel Corp. 9665 Chespapeake Dr.
Suite 325
San Diego 92123
Tel: (619) 292-8926 FAX: (619) 292-0628

Intel Corp.* 4885 Tennessee Avenue
Suite 450
San Diego 92108
Tel: (714) 835-9642 FAX: (714) 956-1114

CALIFORNIA
Intel Corp. 29 Technology Park
Suite 400
Schaeberg 60173
Tel: (404) 449-5641 FAX: (404) 605-9762

ILLINOIS
Intel Corp.* 300 N. Martingale Road
Suite 400
 Schaumberg 60173
Tel: (212) 675-8031 FAX: (212) 708-9762

INDIANA
Intel Corp. 8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623 FAX: (317) 875-9398

NEVADA
Intel Corp. 1930 St. Andrews Drive NE
Suite 200
Cedar Rapids 52402
Tel: (319) 393-1294

KANSAS
Intel Corp. 10000 Century Dr.
Suite 140
Overland Park 66210
Tel: (913) 345-2727 FAX: (913) 345-2076

MARYLAND
Intel Corp.* 5000 Executive Center Dr.
Suite 100
Annapolis Junction 20701
Tel: (301) 206-1696 FAX: (301) 206-3679

MASSACHUSETTS
Intel Corp.* Westford Corp. Center
3 Currie Road
Westford 01886
Tel: (508) 692-3222 FAX: (508) 692-7897

MICHIGAN
Intel Corp. 7071 Orchard Lake Road
Suite 100
West Bloomfield 48322
Tel: (313) 851-8770 FAX: (313) 851-8770

MINNESOTA
Intel Corp. 3530 W. 6th St.
Suite 360
Bloomington 55431
Tel: (612) 699-2697 FAX: (612) 631-6497

MISSOURI
Intel Corp. 4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990 FAX: (314) 291-1491

NEW JERSEY
Intel Corp.* Parkway Office Center
328 Newman Springs Road
Teaneck 07666
Tel: (201) 747-8993 FAX: (201) 747-1095

NEW YORK
Intel Corp.* 10000 Willowbrook
Suite 100
South Trail 11322
Tel: (631) 350-2330 FAX: (631) 346-7036

OHIO
Intel Corp. 5240 Center Park Drive
Suite 250
Dayton 45414
Tel: (513) 890-5350 FAX: (513) 890-8685

OKLAHOMA
Intel Corp. 15254 N.W. Greenbrier Parkway
Bldg B
Beaverton 97005
Tel: (503) 645-8351 FAX: (503) 645-8181

PENNSYLVANIA
Intel Corp.* 455 Pennsylvania Avenue
Suite 250
Fort Washington 19034
Tel: (215) 641-2077 FAX: (215) 641-0750

NORTH CAROLINA
Intel Corp. 8000 Executive Center Dr.
Suite 105
Charlotte 28212
Tel: (704) 535-2236 FAX: (704) 535-2236

OHIO
Intel Corp. 5540 Centerview Dr.
Suite 215
Roosevelt 7706
Tel: (919) 851-9537 FAX: (919) 851-9574

UTAH
Intel Corp. 439 East 4400 South
Suite 104
Murray 84107
Tel: (801) 263-9261 FAX: (801) 263-9261

VIRGINIA
Intel Corp. 1504 Santa Rosa Road
Suite 110
Richmond 23288
Tel: (804) 262-5668 FAX: (804) 516-2270

WASHINGTON
Intel Corp. 150 S. 8th Avenue
Suite 300
Seattle 98104
Tel: (206) 453-8806 FAX: (206) 453-3602

Tel: (414) 796-2115

CANADA
BRITISH COLUMBIA
Intel Semiconductor of Canada, Ltd.
4996 Sandown Way
Suite 302
Burnaby V5G 4L6
Tel: (604) 298-0387 FAX: (604) 298-0387

ONTARIO
Intel Semiconductor of Canada, Ltd.
2650 Queenview Drive
Suite 250
Ottawa K28 8K6
Tel: (613) 829-2171 FAX: (613) 829-9506

Quebec
Intel Semiconductor of Canada, Ltd.
630 St. Jean Boulevard
Pointe Claire H9R 3K2
Tel: (514) 694-9130 FAX: (514) 694-9130

1SALES AND SERVICE OFFICE
*FIELD APPLICATION LOCATION
EUROPEAN SALES OFFICES

DENMARK
Intel Denmark A/S
Glenetvej 61, 3rd Floor
2400 Copenhagen NV
Tel: (45) (31) 19 80 33
TLX: 199677

FINLAND
Intel Finland OY
Rusolaituri 2
00390 Helsinki
Tel: (358) 5 544 644
TLX: 123323

FRANCE
Intel Corporation S.A.R.L.
1 Rue Edouard-BP 303
7804 St. Quentin-en-Yvelines Cedex
Tel: (33) (1) 30 57 70 00
TLX: 690016

WEST GERMANY
Intel Semiconductor GmbH*
Dannenstrasse 1
8016 Feldkirchen bei Muenchen
Tel: (49) 089/99092-0
TLX: 5-23177

Intel Semiconductor GmbH
Hochzeitztor Strasse 5
3000 Hannover 1
Tel: (49) 0511/344051
TLX: 9-20305

Intel Semiconductor GmbH
Abraham Lincoln Strasse 16-18
2600 Hildesheim
Tel: (49) 0161/75605-0
TLX: 4-81683

ISRAEL
Intel Semiconductor Ltd.*
Atzmon Industrial Park-Neve Sharet
P.O. Box 42052
Tel-Aviv 61450
Tel: (972) 03/988080
TLX: 371215

ITALY
Intel Corporation Italia S.p.A.*
Mitarbeiter Palazzo E
20090 Assago
Milano
Tel: (39) (02) 92200950
TLX: 341886

NETHERLANDS
Intel Semiconductor B.V.*
Postbus 46150
3099 CC Rotterdam
Tel: (31) 10.407.11.11
TLX: 22883

NORWAY
Intel Norway A/S
Hvammveien 4-PO Box 92
2013 Sjetten
Tel: (47) (6) 842 420
TLX: 798016

SWITZERLAND
Intel Semiconductor A.G.
Zuerichstrasse
6152 Winkel-Ruili bei Zuerich
Tel: (41) 01/860 62 62
TLX: 85077

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.*
Pipers Way
Swindon, Wilshire SN3 1JR
Tel: (44) (0735) 696000
TLX: 4444473

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA
Bacher Electronics G.m.b.h.
Rotenmuellergasse 26
1120 Wien
Tel: (43) (0222) 83 56 46
TLX: 315522

BELGIUM
Instico Belgium S.A.
Av. des Croix de Guerre 94
1120 Bruxelles
Contactpersonen, 94
1120 Brussels
Tel: (02) (02) 216 01 60
TLX: 64475 or 22290

DENMARK
ITT-Multikomponent
Naverland 29
2600 Glosstrup
Tel: (45) (0) 2 45 66 45
TLX: 840552

FINLAND
OV Fristrons AB
Moliakulku 24A
002110 Helsingi
Tel: (358) (0) 9290292
TLX: 124224

FRANCE
Arnex
Zone Industrielle d’Antony
48, rue de l’Aubepine
BP 103
92164 Antony cedex
Tel: (33) (1) 46 66 21 12
TLX: 250067
Jemyn-Generim
63, rue des Geneaux
Silc 580
94053 Rungis cedex
Tel: (33) (1) 47 76 49 76
TLX: 201585
Metrologie
Tour des Arialles
4, av. Laurent-Cely
92086 Asnieres Cedex
Tel: (33) (1) 47 90 62 40
TLX: 611448

Tekesle-Antronic
Cite des Bruyeres
8, rue Charles Vernet - BP 2
92010 Sevran
Tel: (33) (1) 46 34 75 35
TLX: 20555

WEST GERMANY
Electronic 2000 AG
Stahlhang 12
8000 Muenchen 82
Tel: (49) 0921/40010
TLX: 520561

ITT Multikomponent GmbH
Postfach 1265
Bahnhoftorstrasse 44
D-10004 Berlin
Tel: (49) 06114879
TLX: 7294472

Jemyn GmbH
Im Dachstueck 9
6250 Lithouen
Tel: (49) 09431/508-0
TLX: 41000

Metrologie GmbH
Meigenstrasse 42
8000 Muenchen 71
Tel: (49) 09431/8042-0
TLX: 521318

Prolektron Vertriebs GmbH
Max Planck Strasse 1-3
8072 Dornich
Tel: (49) 09163/2812
TLX: 471903

IRELAND
Micro Marketing Ltd.
Glenageary Office Park
Glanagary
Co. Dublin
Tel: (21) (353) (0) 85 63 25
TLX: 351984

ISRAEL
Electronics Ltd.
11 Rosenzweig Street
P.O.B. 32030
Tel-Aviv 61552
Tel: (972) 03-475151
TLX: 33586

ITALY
Intel
Divisione ITT Industries GmbH
Via Mitanion
Palaizzo E5
20090 Assago (MI)
Tel: (39) 02/4540012
TLX: 313525

Laste Elettronica S.p.A.
V. le Faavo Testa, 126
20092 Cinsello Balsamo
(Tel: (39) 02/4540012
TLX: 323040

Telcom S.r.i.
Via Cà Civilta, 75
20148 Milano
Tel: (39) 341/40098
TLX: 325665

ITT Multicomponents
Viale Milano/ò E5
20090 Assago (MI)
Tel: (39) 02/4540012
TLX: 313525

Silverstar
Via Dei Gracchi 20
20146 Milano
Tel: (39) 341/40098
TLX: 323169

NETHERLANDS
Koning en Hartman Elektrotechnie B.V.
Energieweg 1
2627 AP Delft
Tel: (31) (0) 15/359906
TLX: 382500

NORWAY
Nordkraft Elektronikk (Norge) A/S
Postboks 123
Smadsvingen 4
1364 Halden
Tel: (47) (2) 84 62 10
TLX: 77546

PORTUGAL
ATD Portugal LDA
Rua Dos Lusiticos, 5 Sala B
1300 Lisboa
Tel: (35) (1) 64 80 91
TLX: 61842

UNITED KINGDOM
Accent Electronic Components Ltd.
Jubilee House, Jubilee Road
Leitchworth, Herts SG6 1TL
Tel: (44) (0462) 686666
TLX: 826295

Bytech-Conway Systems
3 The Western Centre
Western Road
Bracknell RG12 1RW
Tel: (44) (0344) 55333
TLX: 847591

Jemyn
Vestry Estate
Oxford Road
Sevenoaks
Kent TN4 1EU
Tel: (44) (0734) 450144
TLX: 95142

MMD
Unit B Southwark Park
Caversham Reading
Berks RG4 GAF
Tel: (44) (0734) 481666
TLX: 846696

Silicon Rapid
Pen House
Denmark Street
High Wycombe
Buckinghamshire HP11 2ER
Tel: (44) (0494) 442266
TLX: 625951

Rapid Systems
Pen House
Denmark Street
High Wycombe
Buckinghamshire HP11 2ER
Tel: (44) (0494) 450244
TLX: 837931

YUGOSLAVIA
H.R. Microelectronics Corp.
2005 de la Cruz Blvd., Ste. 223
Sante Clara, CA 95050
U.S.A.
Tel: (1) (408) 988-9286
TLX: 367452

Rapido Electronic Components
S.p.A.
Villa Deccaria, 8
34133 Trieste
Italy
Tel: (39) (040) 390568
TLX: 460461

*Field Application Location
DOMESTIC SERVICE OFFICES

ALABAMA
*Intel Corp. 5015 Bradford Dr., Suite 2 Huntsville 35805 Tel: (205) 830-4010

ALASKA
Intel Corp. c/o TransAlaska Data Systems 300 Old Steese Hwy., Fairbanks 99707-3120 Tel: (907) 452-4401

ALBERTA
Intel Corp. Tel: (403) 266-2612

ARIZONA
*Intel Corp. 11255 N. 28th Dr. Phoenix 85029 Tel: (602) 975-9900

ARIZONA
*Intel Corp. 3500 N. 28th Dr. Phoenix 85017 Tel: (602) 975-9900

CALIFORNIA
*Intel Corp. 2151 Vanowen St., Ste. 116 North Hollywood 91606 Tel: (818) 704-8500

CALIFORNIA
*Intel Corp. 400 N. Tuscon Avenue Suite 450 Santa Clara 95052 Tel: (408) 970-1700 1-800-421-0386

CALIFORNIA
*Intel Corp. 2700 San Tomas Exp., 2nd Floor Santa Clara 95051 Tel: (408) 976-6086

COLORADO
*Intel Corp. 650 S. Cherry St., Suite 915 Denver 80222 Tel: (303) 321-6086

CONNECTICUT
*Intel Corp. 301 Lee Farm Corporate Park 83 Wooster Heights Rd. Danbury 06810 Tel: (203) 748-3130

FLORIDA
*Intel Corp. 6933 N.E. 6th Way, Ste. 100 Ft. Lauderdale 33309 Tel: (954) 771-0600

GEORGIA
*Intel Corp. 3200 Ponce de Leon Blvd., Ste. 200 Norcross 30092 Tel: (404) 449-0541

HAWAII
*Intel Corp. U.S.I.E.C. Signal Bldg. 1-152 Shafter Pkwy Shafter 96756

ILLINOIS
*Intel Corp. 200 N. Martinique Rd., Ste. 400 Schaumburg 60173 Tel: (312) 605-9631

INDIANA
*Intel Corp. 8777 Purdue Rd., Ste. 125 Indianapolis 46268 Tel: (317) 975-0623

ILNOIS
*Intel Corp. 300 N. Martinique Road Suite 300 Schaumburg 60173 Tel: (708) 756-7070 1-800-421-0386

KANSAS
*Intel Corp. 10995 Cody, Suite 140 Overland Park 66210 Tel: (913) 342-2727

MARYLAND
**Intel Corp. 10010 Junction Dr., Suite 200 Annapolis Junction 20701 Tel: (301) 208-2963 FAX: 301-208-3677

MASSACHUSETTS
**Intel Corp. 3 Carlisle Rd., 2nd Floor Westford 01886 Tel: (978) 690-1060

MICHIGAN
*Intel Corp. 7071 Orchard Lake Rd., Ste. 100 West Bloomfield 48322 Tel: (248) 651-8905

MINNESOTA
**Intel Corp. 5500 W. 89th St., Suite 360 Bloomington 55431 Tel: (952) 835-6722

MISSOURI
**Intel Corp. 15254 N.W. Greenbrier Parkway Building B 79705 Tel: (816) 241-8087

OREGON
**Intel Corp. 3600 W. 28th Ave., Suite 108 Portland 97210 Tel: (503) 658-6888

PENNSYLVANIA
*Intel Corp. 1504 South Road, Ste. 108 Towanda 18848 Tel: (570) 241-8087

WASHINGTON
*Intel Corp. 225 E. Franklin St., Suite 200 Bellevue 98004 Tel: (425) 453-8086

CANADA
Ontario
*Intel Corp. 300 N. Martinique Road Suite 300 Schaumburg 60173 TEL: (708) 756-7070 1-800-421-0386

CANADA
Ontario
Intel Semiconductor of Canada, Ltd. 2650 Queenston Dr., Ste. 250 Mississauga L2C 2B4 Tel: (905) 829-9714 FAX: 905-820-5935

CUSTOMER TRAINING CENTERS

CALIFORNIA
2700 San Tomas Expy. Santa Clara 95051 Tel: (408) 970-1700 1-800-421-0386

ILLINOIS
300 N. Martinique Road Suite 300 Schaumburg 60173 Tel: (708) 756-7070 1-800-421-0386

MASSACHUSETTS
3 Carlisle Rd., 2nd Floor Westford 01886 Tel: (978) 690-1060 1-800-328-0386

MARYLAND
10010 Junction Dr. Suite 300 Annapolis Junction 20701 Tel: (301) 208-2963 1-800-328-0386

SYSTEMS ENGINEERING MANAGERS OFFICES

MINNESOTA
3500 W. 80th Street Suite 360 Bloomington 55431 Tel: (612) 292-8086

NEW YORK
2950 Expressway Dr., South Islandia 11722 Tel: (516) 231-3300

Puerto Rico
*Intel Corp. South Industrial Park P.O. Box 910 Las Piedras 00971 Tel: (809) 733-6616

TEXAS
*Intel Corp. 8815 Dyer St., Suite 225 El Paso 79904 Tel: (915) 751-0186

Utah
*Intel Corp. 1513 Cedar Cliff Dr. Camp Hill 17011 Tel: (717) 761-0060

**Carry-in locations
**Carry-in/in-mail locations

CUSTOMER TRAINING CENTERS

CALIFORNIA
2700 San Tomas Expy. Santa Clara 95051 Tel: (408) 970-1700 1-800-421-0386

ILLINOIS
300 N. Martinique Road Suite 300 Schaumburg 60173 Tel: (708) 756-7070 1-800-421-0386

MASSACHUSETTS
3 Carlisle Rd., 2nd Floor Westford 01886 Tel: (978) 690-1060 1-800-328-0386

MARYLAND
10010 Junction Dr. Suite 300 Annapolis Junction 20701 Tel: (301) 208-2963 1-800-328-0386

SYSTEMS ENGINEERING MANAGERS OFFICES

MINNESOTA
3500 W. 80th Street Suite 360 Bloomington 55431 Tel: (612) 292-8086

NEW YORK
2950 Expressway Dr., South Islandia 11722 Tel: (516) 231-3300

Puerto Rico
*Intel Corp. South Industrial Park P.O. Box 910 Las Piedras 00971 Tel: (809) 733-6616

TEXAS
*Intel Corp. 8815 Dyer St., Suite 225 El Paso 79904 Tel: (915) 751-0186

Utah
*Intel Corp. 1513 Cedar Cliff Dr. Camp Hill 17011 Tel: (717) 761-0060

**Carry-in locations
**Carry-in/in-mail locations
Embedded Applications

The 80960 embedded microprocessor family provides high performance 32-bit RISC-based architecture! Charts, diagrams, technical specifications and architectural information are all great tools for the designer but what is even better is application techniques. Experience shared is the best teacher. From laser printers to production control, these notes discuss hardware and software implementation and present helpful techniques.

Read how Intel’s 8-, 16- and 32-bit embedded controllers and processors are put to work in this collection of application notes. This edition contains over 50 application notes and article reprints on topics including the new 80960 embedded processors, MCS-48, MCS-51 and MCS-96 families. Also covered is the 80186/80188 family and a general section on microcontrollers.