M68HC05AG/AD

HC05

M68HC05 APPLICATIONS GUIDE

MOTOROLA

HC05
M68HC05 APPLICATIONS GUIDE
MOTOROLA

M68HC05 MICROCONTROLLER

Applications Guide

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not authorized for use as components in life support devices or systems intended for surgical implant into the body or intended to support or sustain life. Buyer agrees to notify Motorola of any such intended end use whereupon Motorola shall determine availability and suitability of its product or products for the use intended. Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Employment Opportunity/Affirmative Action Employer.

TABLE OF CONTENTS

Paragraph Number | Title | Page Number

Section 1
Introduction

Section 2
Microcontroller Operation

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

Section 4
Applications

Appendix A
Instruction Set Details

Appendix B
Review Questions

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Concluded)

LIST OF TABLES

SECTION 1
INTRODUCTION

Welcome to the world of microcontrollers!

In this applications guide, we will develop a project using a Motorola MC68HC705C8 microcontroller unit (MCU) in a familiar application — a home thermostat. The MC68HC705C8 is a member of the M68HC05 Family of MCUs. The project will demonstrate only a few of the many possible microcontroller functions that you can use.

This guide assumes that you have no knowledge of microcontrollers and no MCU applications experience.

Section 1 begins with definitions, gives background information, and describes computer systems. An overview of microcontroller applications is also presented and an application project is discussed.

Section 2 describes in detail how microcontrollers operate.

Section 3 contains functional data for the Motorola MC68HC705C8 MCU. This section gives you specific information needed to use this MCU in an application. More information can be found in slightly different form in BR594/D, the *MC68HC705C8 Technical Summary*, which is available separately.

Section 4 shows you how to develop applications and gives you the thermostat project details.

Appendix A provides a detailed description of each instruction in the MC68HC05 instruction set.

Appendix B contains review questions, answers, and explanations.

1.1 DEFINITIONS

The heart of a computer is the central processor unit (CPU). A microprocessor is a CPU on a single chip.

A computer system is a CPU plus peripherals such as input/output (I/O) devices, memory, a program, and a timing reference.

A microcontroller is a very small product that contains many of the functions found in any computer system. A microcontroller uses a microprocessor (as its CPU) as well as memory and peripherals on the same chip.

A microcontroller (MCU) is packaged as a single chip that can be programmed *by the user* with a series of instructions loaded into its memory.

1.2 BACKGROUND

Before MCUs, controllers were hard-wired electronic devices whose operation was determined by the circuits and wires contained within them.

The operation of an MCU-based controller is determined primarily by its program instead of its components and wires. Any function that can be implemented using hard-wired digital integrated circuits (ICs) can also be implemented and performed by an MCU.

As the size and complexity of the devices increase, MCUs become attractive for two reasons:

1. The hard-wired approach requires adding ICs to perform more complex tasks; whereas, MCUs require only a longer program.
2. Microcontrollers are more versatile. Any change in a hard-wired system usually involves replacing ICs and rerouting wires. Most modifications to an MCU system are made simply by changing the program.

MCUs are very useful where many decisions or calculations are required. It is easier to use the computational power of a computer than to use discrete logic.

Microcontrollers are now being used to replace existing designs because they are far simpler to use than conventional IC logic. Since the MCU approach is programmable, many additional features are possible at little or no added cost. Programmability makes possible multiple use of a common piece of hardware since only the control program needs to be changed.

1.3 COMPUTER SYSTEMS DESCRIPTION

Whatever their size, all computer systems consist of the same fundamental parts: CPU, I/O devices, memory, program(s), and a timing reference (clock) as shown in Figure 1-1.

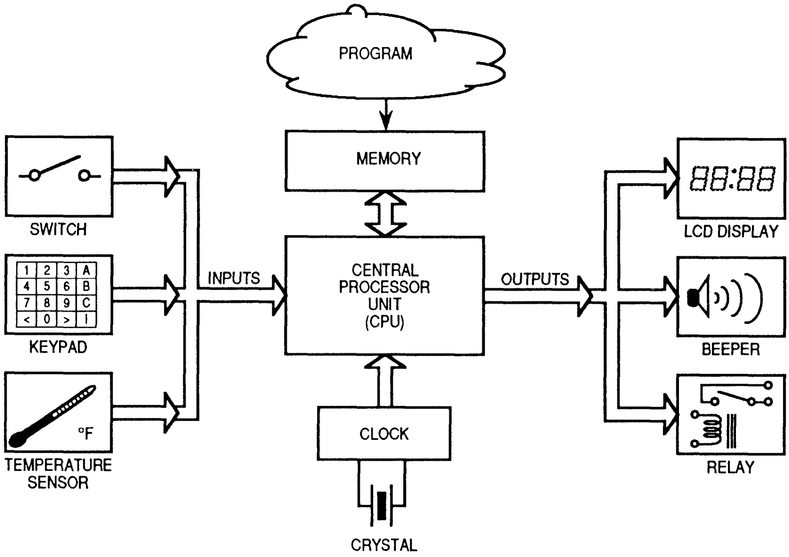


Figure 1-1. A Typical Computer System

The CPU processes information in accordance with a program of instructions and data in a particular language called machine code. The CPU controls all the system operations and provides control signals for enabling and disabling the various peripherals and I/O devices.

Input devices supply information to the MCU from the outside world. Some input devices convert analog signals into digital signals that the MCU can understand and manipulate. Other input devices translate real-world information into the 0 to +5 Vdc signals required by MCUs. Examples of this are a temperature sensor, a switch, a keypad, and a typewriter-style keyboard. A computer system might have one or a number of these input devices.

Output devices are controlled by signals from the MCU. An external interface is required by some output devices to translate the 0 to +5 Vdc MCU levels into different voltage or current levels. Liquid crystal displays, video display terminals, and heating/cooling equipment are examples of output devices.

Memory can store information, including the instructions and data that the CPU uses. The two basic memory types are random access memory (RAM) and read-only memory (ROM).

RAM is used for temporary storage of data and instructions. The computer system can write information into and read information from a RAM in an arbitrary random order. RAM is volatile in that its contents are lost when power is removed.

ROM has data and instructions (a program) stored permanently in it when it is manufactured. The CPU can read information from a ROM but cannot write information into it. ROM information is nonvolatile in that it does not change even when power is removed.

A programmable read-only memory (PROM) is a type of ROM that can be programmed by the user.

An erasable programmable read-only memory (EPROM) is a type of PROM that can be erased by exposing it to ultraviolet light. Once erased, an EPROM may be reprogrammed with new instructions and data.

An OTPROM is a type of EPROM that is manufactured in an inexpensive plastic package. Since the plastic package is opaque to ultraviolet light, an OTPROM can be programmed only once.

Like ROM, PROM, EPROM, and OTPROM are nonvolatile types of memory.

The program contains instructions and data. The computer system uses the program to perform some desired process(es).

The computer clock is used for timing and sequencing the various operations. A crystal is usually used to provide the reference frequency for the clock.

1.4 MICROCONTROLLER APPLICATIONS OVERVIEW

The development of a new microcontroller application is limited only by skill and imagination, since the elements of a microcontroller system are easily assembled. MCU applications generally allow many new functions that make process control simpler and more powerful, often at reduced cost.

Many applications require analog inputs and outputs. The resulting system is the equivalent of a traditional analog controller with a number of control loops. Control loops regulate an output as a function of one or more inputs. Control loops are illustrated in the flowchart of Figure 1-2.

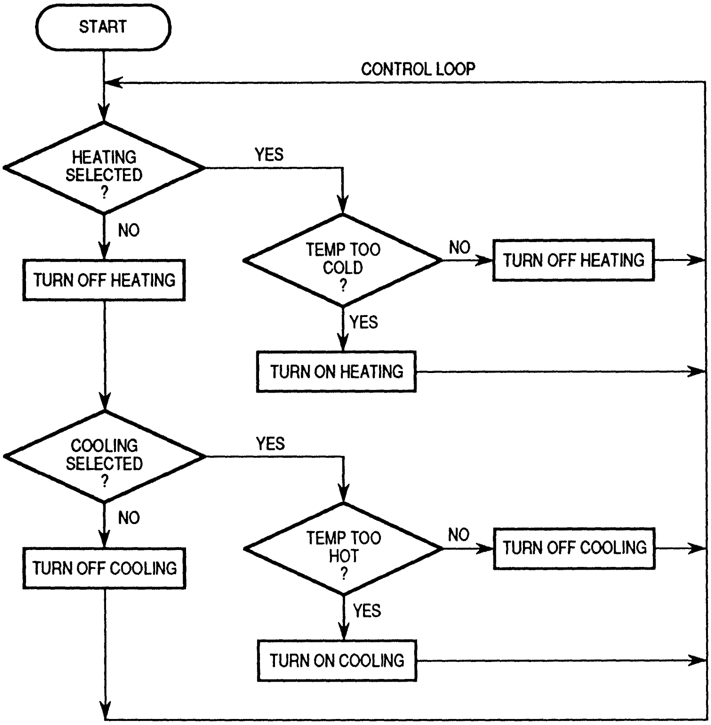


Figure 1-2. A Temperature Control Flowchart

Some applications have costly sensors and control mechanisms. The cost of the sensors required for input and the cost of the control devices required for output are usually much greater than the cost of a standard MCU.

The advantage of an MCU system is the use of software to replace complex and expensive hardware previously required. The cost of the software is a tradeoff against the cost of the additional hardware and the space it requires.

Programming allows use of complex functions that could not easily be accomplished with hard-wired devices. Changes in functions can be made and programs can be improved or replaced with few or no hardware changes.

1.5 PROJECT DESCRIPTION

A basic thermostat controller was chosen for this project because it should be familiar to all readers and because it includes the fundamental elements common to all MCU applications. Figure 1-3 illustrates a home thermostat controller that can control both heating and air conditioning.

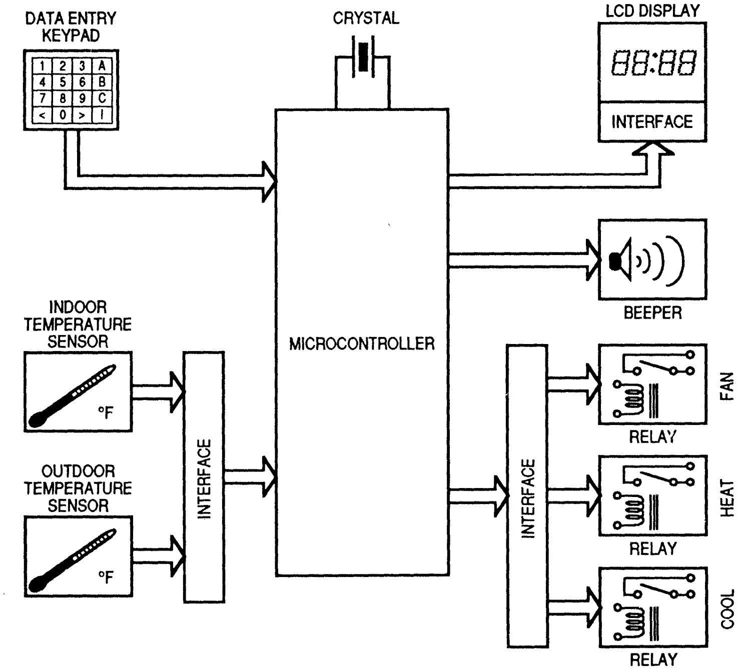


Figure 1-3. Thermostat Project Block Diagram

Since the thermostat is based on an MCU, complex functions can be added. The thermostat could include a timed setback feature that allows specifying certain times of the day when there will be reduced demand for heating or air conditioning, thus giving some energy savings. A more unusual feature would be to measure the outdoor temperature and control the indoor-to-outdoor temperature difference. This would be very difficult to accomplish with a conventional electromechanical thermostat.

The four fundamental elements of this system are inputs, outputs, time, and a microcontroller to tie the other elements together. The inputs include push-buttons (a keypad) to enter time and temperature information into the MCU and sensors to measure the indoor and outdoor temperatures. Outputs include a display to show system conditions and signals to the interfaces that control the heating and air conditioning equipment. Time is derived from a crystal connected to the MCU. As we will see later, this crystal would be used by the CPU even if the application did not have time-of-day requirements. A program controls the entire operation of the thermostat. Section 4 of this manual contains project details.

SECTION 2
MICROCONTROLLER OPERATION

A microcontroller unit (MCU) is a complete computer system on a single silicon chip. In a great many controller applications, the MCU can satisfy all system requirements with no additional integrated circuits (ICs). Due to very low cost and a high degree of flexibility, these powerful new MCU devices are finding their way into many applications that were previously accomplished with combinational logic or even by mechanical means. As a result, there are many experienced engineers who need to become familiar with the function and application of Motorola MCUs. This section, which is specifically designed for those engineers, is also a good reference for engineers who are familiar with MCUs from some other manufacturer.

The MCU block in the thermostat block diagram of Figure 1-3 can be expanded as shown in Figure 2-1 to show the functional blocks within the MCU. The

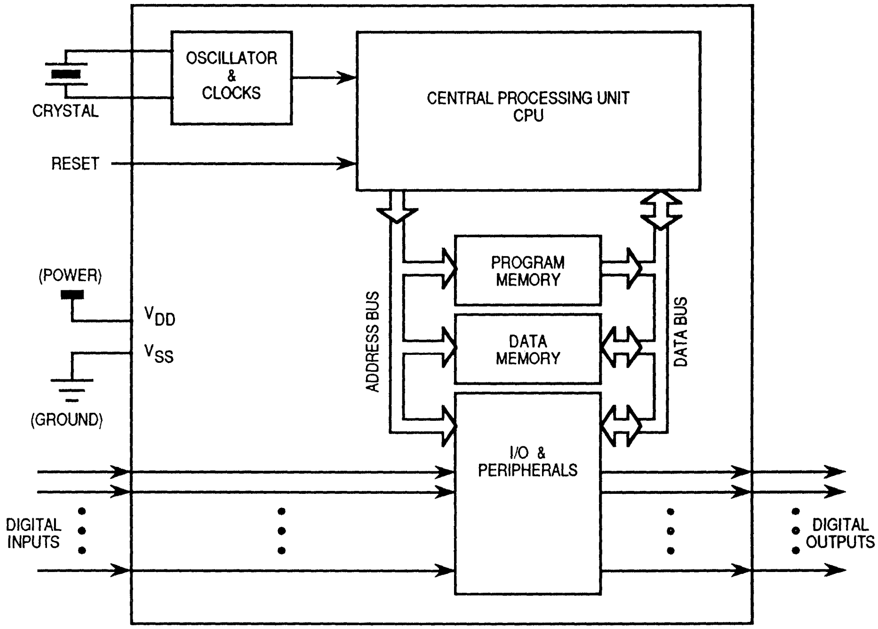


Figure 2-1. MCU Expanded Block Diagram

CPU block is the central element of a digital binary computer much like the mainframe computers used in business except that it is much smaller. The goal of this section is to study the internal operation of this CPU and how it interacts with the other functional blocks within the MCU. Although this discussion is based on a relatively simple CPU, the principles apply to even the most powerful mainframe computers.

The CPU is a system of simple logic elements and buses that can sequentially interpret and execute a finite set of instructions. Starting from a specific address in memory after reset, the CPU mindlessly fetches and executes one simple instruction after another. Each instruction is composed of several even simpler steps. The small substeps comprising each instruction are determined by the wiring within the CPU. The transistors, logic gates, and buses which comprise the CPU are called hardware. The instructions the CPU follows to accomplish an application task are determined by an end user or design engineer and are called a software program.

Before we can get into the discussion of the internal operations of the CPU, some basic concepts must be understood. The following paragraphs discuss numbering systems and special codes used by computers.

2.1 NUMBER SYSTEMS

Computers work best with information in a different form than people use. Humans typically work in the base 10 (decimal) numbering system (probably because we have ten fingers). Digital binary computers work in the base 2 (binary) numbering system because this allows all information to be represented by sets of digits, which can only be zeros or ones. In turn, a one or zero can be represented by the presence or absence of a logic voltage on a signal line or the on and off states of a simple switch.

In decimal (base 10) numbers, the weight of each digit is ten times as great as the digit immediately to its right. The rightmost digit of a decimal integer is the ones place, the digit to its left is the tens digit, and so on. In binary (base 2) numbers, the weight of each digit is two times as great as the digit immediately to its right. The rightmost digit of the binary integer is the ones digit, the next digit to the left is the twos digit, next is the fours digit, then the eights digit, and so on.

Although computers are quite comfortable working with binary numbers of 8, 16, or even 32 binary digits, humans find it very inconvenient to work with so many digits at a time. The base 16 (hexadecimal) numbering system offers a practical compromise. One hexadecimal digit can exactly represent four binary digits, thus, an 8-bit binary number can be expressed by two hexadecimal digits.

The correspondence between a hexadecimal digit and the four binary digits it represents is simple enough that humans who work with computers easily learn to mentally translate between the two. In hexadecimal (base 16) numbers, the weight of each digit is 16 times as great as the digit immediately to its right. The rightmost digit of a hexadecimal integer is the ones place, the digit to its left is the sixteens digit, and so on.

Table 2-1 demonstrates the relationship between the decimal, binary, and hexadecimal representations of values. These three different numbering systems are just different ways to represent the same physical quantities.

Table 2-1. Decimal, Binary, and Hexadecimal Equivalents

Base 10 Decimal	Base 2 Binary	Base 16 Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	10
17	0001 0001	11
100	0110 0100	64
255	1111 1111	FF
1024	0100 0000 0000	400
65,535	1111 1111 1111 1111	FFFF

The letters A through F are used to represent the hexadecimal values corresponding to 10 through 15 because each hexadecimal digit can represent 16 different quantities; whereas, our customary numbers only include the 10 unique symbols (0 through 9). Thus, some other single-digit symbols had to be used to represent the hexadecimal values for 10 through 15.

To avoid confusion about whether a number is decimal or hexadecimal, hexadecimal numbers are preceded by the $ symbol. For example, 64 means decimal "sixty-four"; whereas, $64 means hexadecimal "six-four", which is equivalent to decimal 100. Some other computer manufacturers follow hexadecimal values with a capital H (as in 64H).

Hexadecimal is a good way to express and discuss numeric information processed by computers because it is easy for people to mentally convert between hexadecimal digits and their 4-bit binary equivalent. The hexadecimal notation is much more compact than binary while maintaining the binary connotations.

2.2 COMPUTER CODES

Computers must handle many kinds of information other than just numbers. Text (alphanumeric characters) and instructions must be encoded in such a way that the computer can understand this information. The most common code for text information is the American Standard Code for Information Interchange (or ASCII). The ASCII code establishes a widely accepted correlation between alphanumeric characters and specific binary values. Using the ASCII code, $41 corresponds to capital A, $20 corresponds to a space character, etc. The ASCII code translates characters to 7-bit binary codes, but in practice the information is most often conveyed as 8-bit characters with the most significant bit equal to zero. This standard code allows equipment made by various manufacturers to communicate because all of the machines use this same code.

Computers use another code to give instructions to the CPU. This code is called an operation code or opcode. Each opcode instructs the CPU to execute a very specific sequence of steps that together accomplish an intended operation. Computers from different manufacturers use different sets of opcodes because these opcodes are internally hard-wired in the CPU logic. The instruction set for a specific CPU is the set of all opcodes that the CPU knows how to execute. Even though the opcodes differ from one computer to another, all digital binary computers perform the same kinds of basic tasks in similar ways. The CPU in the MC68HC05 MCU can understand 62 basic instructions. Some of these basic instructions have several slight variations, each requiring a separate opcode. The instruction set of the MC68HC05 includes 210 unique instruction opcodes. We will discuss how the CPU actually executes instructions a little later in this section after a few more basic concepts have been presented.

An opcode such as $4C is understood by the CPU, but it is not very meaningful to a human. To solve this problem, a system of mnemonic instruction formats is used. The $4C opcode corresponds to the INCA mnemonic, which is read "increment accumulator." Although there is printed information to show the correlation between mnemonic instructions and the opcodes they represent, this information is seldom used by a programmer because the translation process is automatically handled by a separate computer program called an assembler. An assembler is a program that converts a program written in mnemonics into a list of machine codes (opcodes) that can be used by a CPU.

An engineer develops a set of instructions for the computer in mnemonic form and then uses an assembler to translate these instructions into opcodes that the CPU can understand. We will discuss instructions, writing programs, and assemblers later in this applications guide, but you should understand that people prepare instructions for a computer in mnemonic form and the computer understands only opcodes; thus, a translation step is required to change the mnemonics to opcodes, and this is the function of the assembler.

Before leaving this discussion of number systems and codes, we will look at two additional codes you may have heard about. Octal (base 8) notation was used for some early computer work but is seldom used today. Octal notation uses the numbers 0 through 7 to represent sets of three binary digits in the same way hexadecimal is used to represent sets of four binary digits. The octal system had the advantage of using customary number symbols (unlike the hexadecimal symbols A through F discussed earlier).

Two disadvantages caused octal to be abandoned for the hexadecimal notation used today. First of all, most computers use 4, 8, 16, or 32 bits per word; these words do not break down nicely into sets of three bits. (Some early computers used 12-bit words which did break down into four sets of three bits each.) The second problem was that octal is not as compact as hexadecimal. For example, the ASCII value for capital A is 1000001_2 in binary, 41_{16} in hexadecimal, and 101_8 in octal. When a human is talking about the ASCII value for A, it is easier to say "four-one" than it is to say "one-zero-one." When mentally translating from hexadecimal to binary, it is easy to convert each hexadecimal digit into four binary bits. It is more difficult to make the octal-to-binary translation because you have to remember to throw away the leading zero of the first group of three binary bits. You probably had to think twice about that last statement, and that is exactly the point.

Binary coded decimal (BCD) is a hybrid notation used to express decimal values in binary form. BCD uses four binary bits to represent each decimal digit. Since four binary digits can express 16 different physical quantities, there will be six bit-value combinations that are considered invalid (specifically, the hexadecimal values A through F). Values are kept in pseudo-decimal form during calculations.

When the computer does a BCD add operation, it performs a binary addition and then adjusts the result back to BCD form. As a simple example, consider the BCD addition of $9_{10}+1_{10}=10_{10}$. The computer adds $0000\ 1001_2+0000\ 0001_2=0000\ 1010_2$, but 1010_2 is equivalent to A_{16}, which is not a valid BCD value. When the computer finishes the calculation, a check is performed to see if the result is still a valid BCD value. If there was any carry from one BCD digit to another or if there was any invalid code, a sequence of steps would be performed to correct the result to proper BCD form ($0000\ 1010_2$ is corrected to $0001\ 0000_2$ (BCD 10) in this example).

In most cases, it is inefficient to use BCD notation in computer calculations. It is better to change from decimal to binary as information is entered, do all computer calculations in binary, and change the binary result back to BCD or decimal as needed for display. First, not all computers are capable of doing BCD calculations because they need a digit-to-digit carry indicator which is not present on all computers (though Motorola MCUs do have this half-carry indicator). Secondly, forcing the computer to emulate human behavior is inherently less efficient than allowing the computer to work in its native binary system.

2.3 COMPUTER MEMORY

Before the operation of the CPU can be discussed in detail, some conceptual knowledge of computer memory is required. In many beginning programming classes, memory is presented as being similar to a matrix of pigeon holes where you can save messages and other information. The pigeon holes we are referring to are like the mailboxes in a large apartment building. This is a good analogy but needs a little refinement if it is to be used to explain the inner workings of a CPU. We will confine our discussion to an 8-bit CPU so that we can be very specific.

In an 8-bit CPU, each pigeon hole (or mailbox) can be thought of as containing a set of eight on/off switches (eight bits of data are called a byte of data). Unlike a pigeon hole, you cannot fit more information in by writing smaller, and there is no such thing as an empty pigeon hole (though the contents of a memory location can be unknown or undefined at a given time). The switches would be in a row where each switch would represent a single binary digit.

A binary one corresponds to the switch being on, and a binary zero corresponds to the switch being off. Each pigeon hole (memory location) has a unique address so that information can be stored and reliably retrieved.

2.3.1 Computer Architecture

Motorola M68HC05 and M68HC11 8-bit MCUs have a specific organization which is called a Von Neumann architecture after an American mathematician of the same name. In this architecture, a CPU and a memory array are interconnected by an address bus and a data bus. The address bus is used to identify which pigeon hole is being accessed, and the data bus is used to convey information either from the CPU to the memory location (pigeon hole) or from the memory location to the CPU.

In the Motorola implementation of this architecture, there are a few special pigeon holes (called CPU registers) inside the CPU, which act as a small scratch pad and control panel for the CPU. These CPU registers are similar to memory in that information can be written into them and remembered. However, it is important to remember that these registers are directly wired into the CPU and are not part of the addressable memory available to the CPU.

All information (other than the CPU registers) accessible to the CPU is envisioned (by the CPU) to be in a single row of several thousand pigeon holes. This organization is sometimes called a 'memory-mapped I/O' system because the CPU treats all memory locations alike whether they contain program instructions, variable data, or input-output (I/O) controls. There are other computer architectures, but this applications guide is not intended to explore these variations. Fortunately, the Motorola architecture we are discussing is one of the easiest to understand and use. This architecture encompasses the most important concepts of digital binary computers; thus, the information presented in this applications guide will be applicable even if you go on to study other architectures.

The number of wires in the address bus determines the total possible number of pigeon holes; the number of wires in the data bus determines the amount of information that can be stored in each pigeon hole. In the MC68HC705C8, the address bus is 13 bits, making a maximum of 8192_{10} separate pigeon holes (in MCU jargon you would say this CPU can access 8K locations). Since the data bus in the MC68HC705C8 is eight bits, each pigeon hole can hold one byte of information. One byte is eight binary digits, or two hexadecimal digits, or one ASCII character, or a decimal value from 0 to 255.

2.3.2 CPU Registers

Different CPUs have different sets of CPU registers. The differences are primarily the number and size of the registers. Figure 2-2 shows the CPU registers found in an M68HC05. While this is a relatively simple set of CPU registers, it is representative of all types of CPU registers and can be used to explain all of the fundamental concepts.

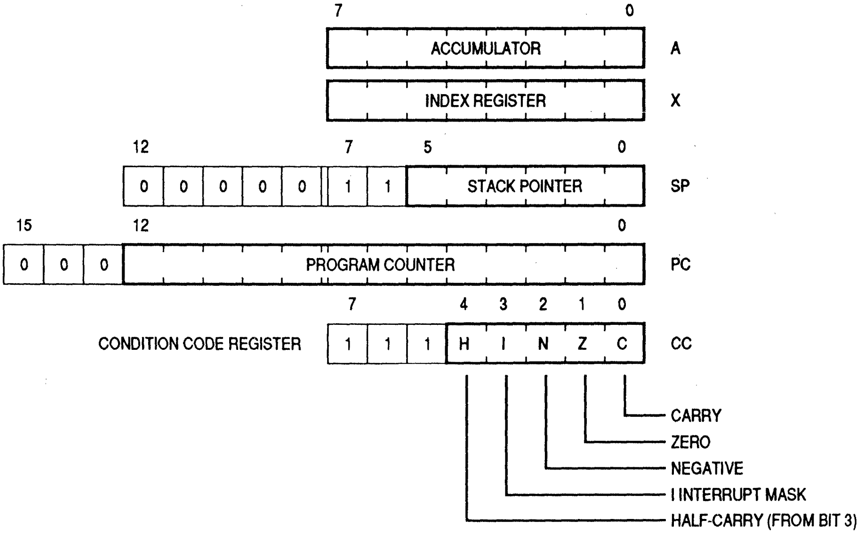


Figure 2-2. M68HC05 CPU Registers

The A register, an 8-bit scratch-pad register, is also called an accumulator because it is often used to hold one of the operands or the result of an arithmetic operation.

The X register is an 8-bit index register, which can also serve as a simple scratch pad. The main purpose of an index register is to point at an area in memory where the CPU will load (read) or store (write) information. Sometimes an index register is called a pointer register. We will learn more about index registers when we discuss indexed addressing modes.

The program counter (PC) register is used by the CPU to keep track of the address of the next instruction to be executed. When the CPU is reset (starts up), the PC is loaded from a specific pair of memory locations called the reset vector. The reset vector locations contain the address of the first instruction to be executed by the CPU. As instructions are executed, logic in the CPU increments the PC such that it always points to the next piece of information that the CPU will need. The number of bits in the PC exactly matches the number of wires in the address bus. This determines the total potentially available memory space that can be accessed by a CPU. In the case of an MC68HC705C8, the PC is 13 bits long; therefore, its CPU can access up to 8K bytes (8192) of memory. Values for this register are expressed as four hexadecimal digits where the upper-order three bits of the corresponding 16-bit binary address are always zero.

The condition code (CC) register is an 8-bit register holding status indicators that reflect the result of some prior CPU operation. The three high-order bits of this register are not used and always stay at logic one. Branch instructions use these status bits to make simple either/or decisions.

The stack pointer (SP) is used as a pointer to the next available location in a last-in-first-out (LIFO) stack. The stack can be thought of as a pile of cards, each holding a single byte of information. At any given time, the CPU can put a card on top of the stack or take a card off the stack. Cards within the stack cannot be used unless all the cards piled on top are removed first. The CPU accomplishes this stack effect by way of the SP. The SP points to a memory location (pigeon hole), which is thought of as the next available card. When the CPU pushes a piece of data onto the stack, the data value is written into the pigeon hole pointed to by the SP; the SP is then decremented so it points at the next previous memory location (pigeon hole). When the CPU pulls a piece of data off the stack, the SP is incremented so it points at the most recently used pigeon hole, and the data value is read from that pigeon hole. When the CPU is first started up or after a reset stack pointer (RSP) instruction, the SP points to a specific memory location in RAM (a certain pigeon hole).

2.3.3 Memory Uses

The computer memory holds all information needed by the computer for instructions, variable data, and even I/O status and controls. Some memory locations contain fixed data like the instructions for the CPU and tables of constant data. This information is typically held in a read-only memory (ROM) although there is no special requirement that this information has to be located in ROM. A second type of information used by computers is variable information that changes often during the operation of the system. This type of data is typically held in a read-write random-access memory (RAM). Information can be read from or written to various locations in RAM in an arbitrary random order. A third type of information found in a computer system is I/O status and control information. This type of memory location allows the computer system to get information to or from the outside world. This type of memory location is unusual because the information can be sensed and/or changed by something other than the CPU.

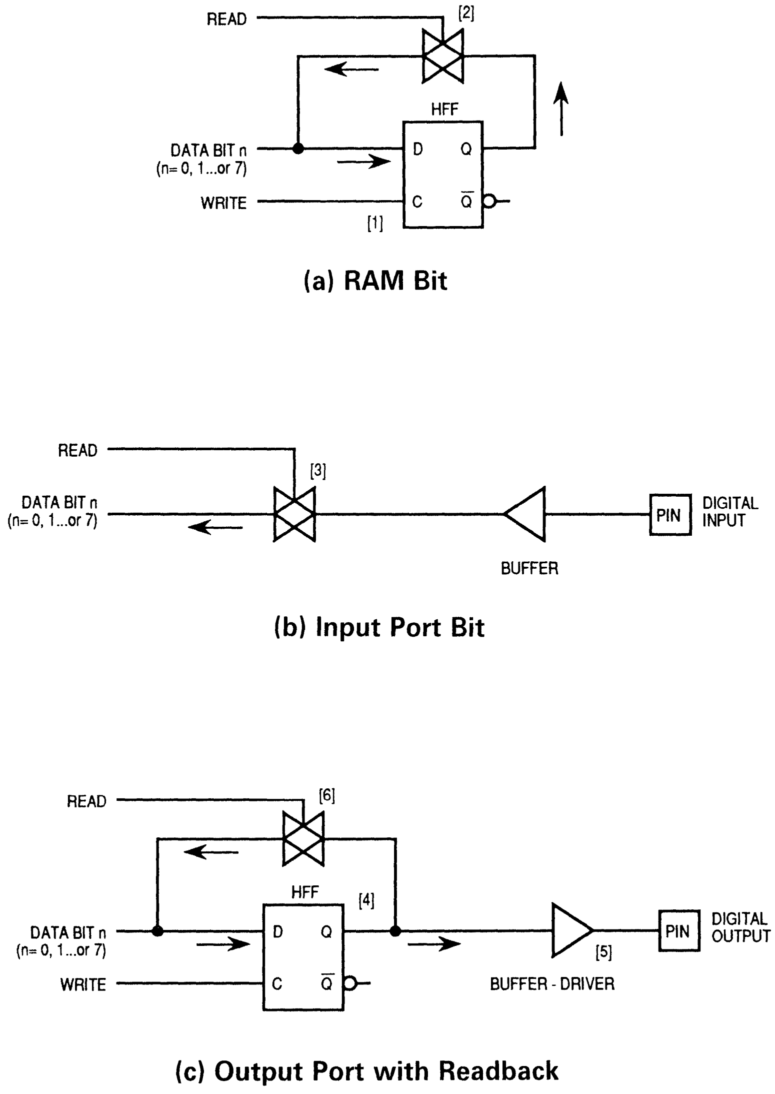
The simplest kind of I/O memory locations are a simple input port and a simple output port. In an 8-bit MCU, a simple input port would consist of eight pins that can be read by the CPU. A simple output port would consist of eight pins that the CPU can control (write to). In practice, a simple output port location is usually implemented with eight latches and feedback paths that allow the CPU to read back what was previously written to the address of the output port.

Figure 2-3 shows the equivalent circuit for one bit of RAM, one bit of an input port, and one bit of a typical output port having readback capability. In a real MCU, this circuit would be repeated eight times to make a single 8-bit RAM location, input port, or output port.

When the CPU stores a value to the address that corresponds to the RAM bit in Figure 2-3 (a), the WRITE signal is activated to latch the data from the data bus line into the flip-flop [1]. This latch is static and remembers the value written until a new value is written to this location (or power is removed). When the CPU reads the address of this RAM bit, the READ signal is activated, which enables the multiplexer at [2]. This multiplexer couples the data from the output of the flip-flop into the data bus line. In a real MCU, RAM bits are actually much simpler than shown here, but they are functionally equivalent to this circuit.

When the CPU reads the address of the input port shown in Figure 2-3 (b), the READ signal is activated, which enables the multiplexer at [3]. The multiplexer couples the buffered data from the pin onto the data bus line. A write to this address would have no meaning.

When the CPU stores a value to the address that corresponds to the output port in Figure 2-3 (c), the WRITE signal is activated to latch the data from the data bus line into the flip-flop [4]. The output of this latch, which is buffered by the buffer driver at [5], appears as a digital level on the output pin. When the CPU reads the address of this output port, the READ signal is activated, which enables the multiplexer at [6]. This multiplexer couples the data from the output of the flip-flop onto the data bus line.



(a) RAM Bit

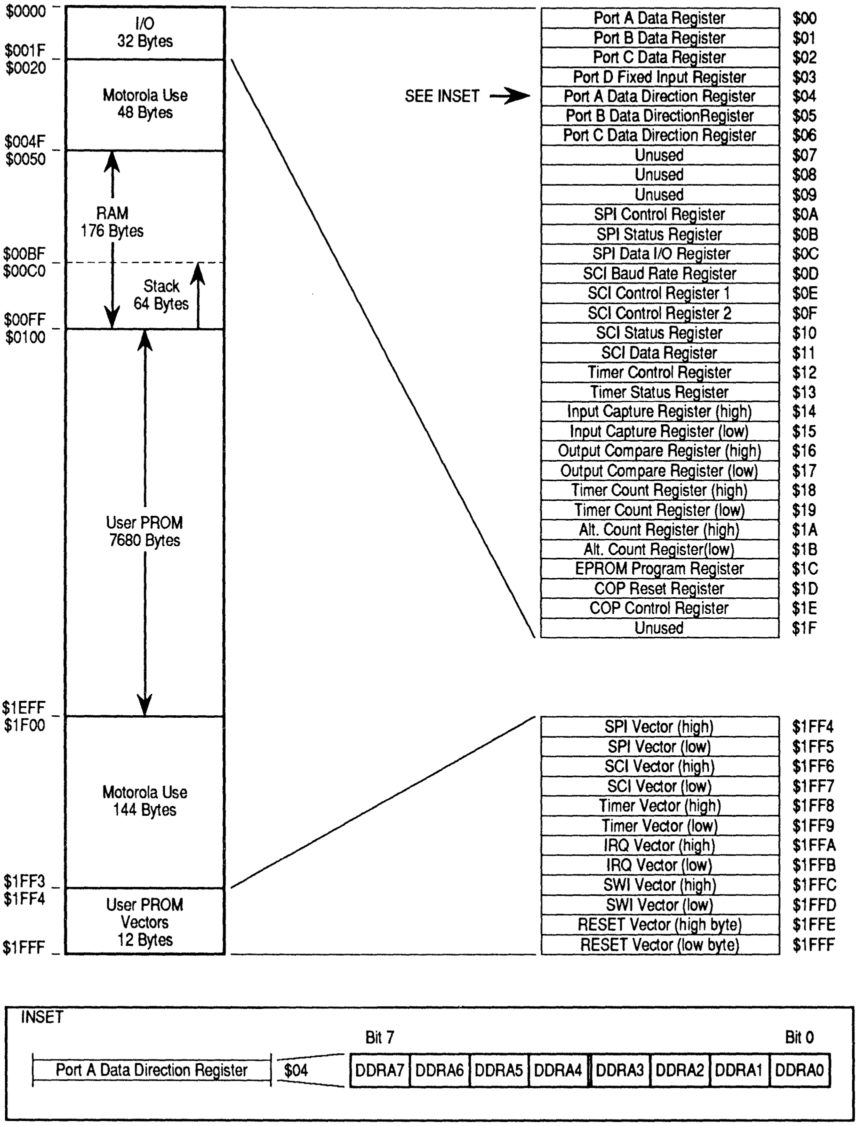
(b) Input Port Bit

(c) Output Port with Readback

Figure 2-3. Memory and I/O Circuitry

2.3.4 Memory Maps

Since there are several thousand memory locations in the MCU system, it is important to have a convenient way to track locations. A memory map is a pictorial representation of the total MCU memory space. Figure 2-4 is a typical memory map showing a subset of the memory resources in the



Port A Data Register	$00
Port B Data Register	$01
Port C Data Register	$02
Port D Fixed Input Register	$03
Port A Data Direction Register	$04
Port B Data DirectionRegister	$05
Port C Data Direction Register	$06
Unused	$07
Unused	$08
Unused	$09
SPI Control Register	$0A
SPI Status Register	$0B
SPI Data I/O Register	$0C
SCI Baud Rate Register	$0D
SCI Control Register 1	$0E
SCI Control Register 2	$0F
SCI Status Register	$10
SCI Data Register	$11
Timer Control Register	$12
Timer Status Register	$13
Input Capture Register (high)	$14
Input Capture Register (low)	$15
Output Compare Register (high)	$16
Output Compare Register (low)	$17
Timer Count Register (high)	$18
Timer Count Register (low)	$19
Alt. Count Register (high)	$1A
Alt. Count Register(low)	$1B
EPROM Program Register	$1C
COP Reset Register	$1D
COP Control Register	$1E
Unused	$1F

SPI Vector (high)	$1FF4
SPI Vector (low)	$1FF5
SCI Vector (high)	$1FF6
SCI Vector (low)	$1FF7
Timer Vector (high)	$1FF8
Timer Vector (low)	$1FF9
IRQ Vector (high)	$1FFA
IRQ Vector (low)	$1FFB
SWI Vector (high)	$1FFC
SWI Vector (low)	$1FFD
RESET Vector (high byte)	$1FFE
RESET Vector (low byte)	$1FFF

Port A Data Direction Register	$04	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0

Figure 2-4. Typical Memory Map

MC68HC705C8. Some memory areas (reserved for Motorola use) were purposely left out of this figure to make it easier to understand. The complete version of this memory map can be found in the Figure 3-5.

The four-digit hexadecimal values along the left edge of Figure 2-4 are addresses beginning with $0000 at the top and increasing to $1FFF at the bottom. $0000 corresponds to the first memory location selected (when the CPU drives all address lines of the internal address bus to logic zero). $1FFF corresponds to the last memory location selected (when the CPU drives all 13 address lines of the internal address bus to logic one). The labels within the vertical rectangle identify what kind of memory (RAM, PROM, I/O registers, etc.) resides in a particular area of memory.

Some areas, such as I/O registers, need to be shown in more detail because it is important to know the names of each individual location. The vertical rectangle can be interpreted as a row of 8192 pigeon holes (memory locations). Each of these 8192 memory locations contains eight bits of data as shown in the inset in Figure 2-4.

The first 256 memory locations ($0000–$00FF) can be accessed with the direct addressing mode of many CPU instructions. In this addressing mode, the CPU assumes that the upper two hexadecimal digits of address are always zeros; thus, only the two low-order digits of the address need to be explicitly given in the instruction. All on-chip I/O registers and 176 bytes of RAM are located in the $0000–$00FF area of memory. In the memory map (Figure 2-4), the expansion of the I/O area of memory identifies each register location with the two low-order digits of its address rather than the full four-digit address. For example, the two-digit hexadecimal value $00 appears to the right of the port A data register, which is actually located at address $0000 in the memory map.

Now that we have some background knowledge of computer memory, we can continue with our discussion of the CPU.

2.4 TIMING

A high-frequency clock source (typically derived from a crystal connected to the MCU) is used to control the sequencing of CPU instructions. Typical MCUs divide the basic crystal frequency by two or more to arrive at a bus-rate clock. Each memory read or write takes one bus-rate clock cycle. In the case of the MC68HC705C8 MCU, a 4-MHz (maximum) crystal oscillator clock is divided by two to arrive at a 2-MHz (maximum) internal processor clock. Each substep of an instruction takes one cycle of this internal processor clock (500 ns). Most instructions take two to five of these substeps; thus, the CPU is capable of executing about 500,000 instructions every second.

2.5 PROGRAMMING

At this point, we will write a short program in mnemonic form, translate it into machine code, and discuss how the CPU would execute the program. This exercise will provide insight into the internal operation of the CPU and computers in general. The instruction set explanations and the process of writing programs will be more understandable with this background.

Our program will read the state of a switch connected to an input pin. When the switch is closed, the program will cause an LED connected to an output pin to light for about one second and then go out. The LED will not light again until the switch has been released and closed again. The length of time the switch is held closed will not affect the length of time the LED is lighted.

Although this program is very simple, it demonstrates the most common elements of any MCU application program. First, it demonstrates how a program can sense input signals such as switch closures. Second, this is an example of a program controlling an output signal. Third, the LED on-time of about one second demonstrates one way a program can be used to measure real time. Because the algorithm is sufficiently complicated, it cannot be accomplished in a trivial manner with discrete components (at minimum, a one-shot IC with external timing components would be required). This example demonstrates that an MCU and a user-defined program (software) can replace complex circuits.

2.5.1 Flowchart

Figure 2-5 is a flowchart of the example program. Flowcharts are often used as a planning tool for writing software programs because they show the function and flow of the program under development. The importance of notes, comments, and documentation for software cannot be overemphasized. Just as you would not consider a circuit-board design complete until there is a schematic diagram, parts list, and assembly drawing, you should not consider a program complete until there is a commented listing and a comprehensive explanation of the program such as a flowchart.

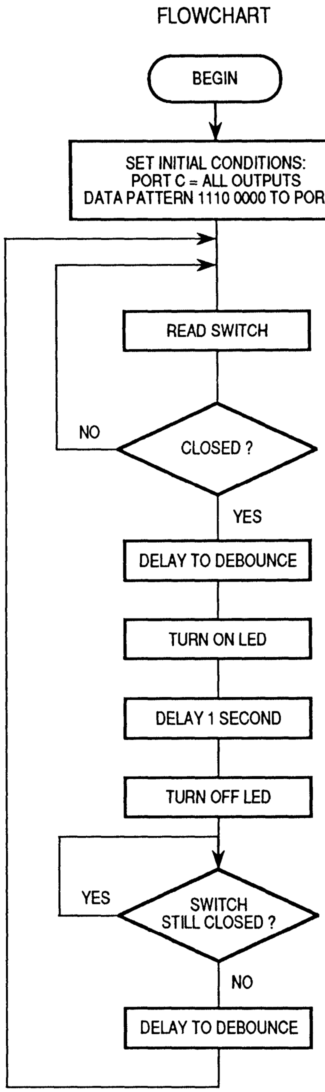
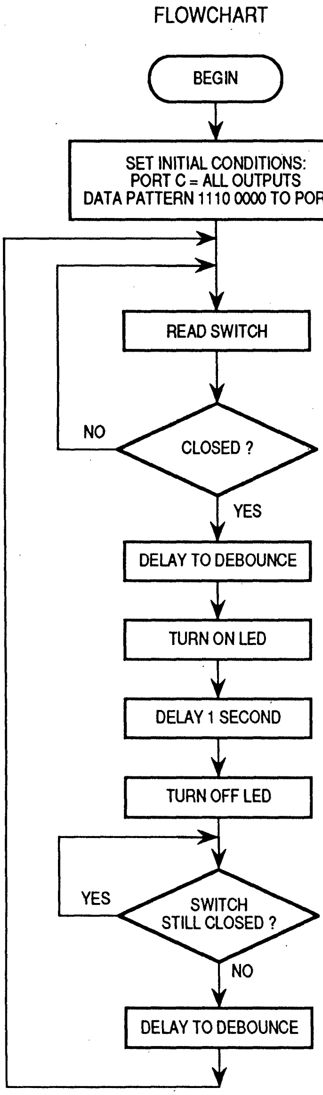


Figure 2-5. Example Flowchart

2.5.2 Mnemonic Source Code

Once the flowchart or plan is completed, the programmer develops a series of assembly language instructions to accomplish the function(s) called for in each block of the plan. The programmer is limited to selecting instructions from the instruction set for the CPU being used (in this case the MC68HC05).

The programmer writes instructions in a mnemonic form which is easy to understand. Figure 2-6 shows the mnemonic source code next to the flowchart of our example program so you can see what CPU instructions are used to accomplish each block of the flowchart. The meanings of the mnemonics used in the right side of Figure 2-6 can be found in Appendix A.



MNEMONIC PROGRAM

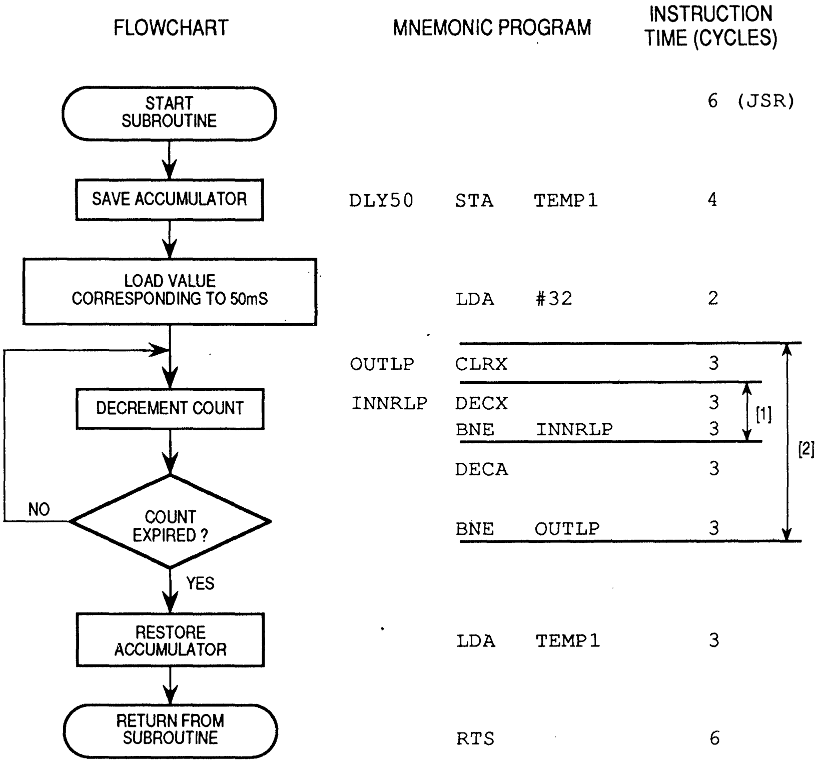
```
INIT    LDA    #$FF
        STA    DDRC
        LDA    #$E0
        STA    PORTC

TOP     LDA    PORTB

        BPL    TOP

        JSR    DLY50

        BCLR   6,PORTC
        LDA    #20
DLYLP   JSR    DLY50
        DECA
        BNE    DLYLP
        BSET   6,PORTC

OFFLP   BRSET  7,PORTB,OFFLP

        JSR    DLY50
        BRA    TOP
```

Figure 2-6. Flowchart and Mnemonics

During development of the program instructions, it was noticed that a time delay was needed in three places. A subroutine was developed that would generate a 50-ms delay. This subroutine was used directly in two places (for switch debouncing) and made the one-second delay easier to produce. To keep this figure simple, the comments that would usually be included within the source program for documentation are omitted. The comments will be shown in the complete assembly listing in Figure 2-8.

2.5.3 Software Delay Program

Figure 2-7 shows an expanded flowchart of the 50-ms delay subroutine. A subroutine is a relatively small program which performs some commonly required function. Even if the function needs to be performed many times



```
                              6 (JSR)
DLY50   STA     TEMP1         4
        LDA     #32           2
OUTLP   CLRX                  3
INNRLP  DECX                  3
        BNE     INNRLP        3
        DECA                  3
        BNE     OUTLP         3
        LDA     TEMP1         3
        RTS                   6
```

[1] - INNRLP is executed 256 times on each pass through outer loop.

[2] - OUTLP is executed 32 times.

Figure 2-7. Delay Routine Flowchart and Mnemonics

in the course of a program, the subroutine only has to be written once. Each place where this function is needed, the programmer would call the subroutine with a branch-to-subroutine (BSR) or jump-to-subroutine (JSR) instruction.

Before starting to execute the instructions in the subroutine, the address of the instruction which follows the JSR (or BSR) is automatically stored in temporary RAM memory locations. When the CPU finishes executing the instructions within the subroutine, a return-from-subroutine (RTS) instruction is performed as the last instruction in the subroutine. The RTS instruction causes the CPU to recover the previously saved return address; thus, the CPU continues the program with the instruction following the JSR (or BSR) instruction that originally called the subroutine.

The delay routine of Figure 2-7 involves an inner loop (INNRLP) within another loop (OUTLP). The inner loop consists of two instructions executed 256 times before X reaches $00 and the BNE branch condition fails. This amounts to six cycles at 1 μs/cycle times 256, which equals 1.536 ms for the inner loop. The outer loop executes 32 times. The total execution time for the outer loop is 32(1536+9) or 32(1545)=49.44 ms. The miscellaneous instructions in this routine other than those in the outer loop total 21 cycles; thus, the total time required to execute the DLY50 routine is 49.461 ms, including the time required for the JSR instruction that calls DLY50.

The 16-bit timer system in the MC68HC705C8 can also be used to measure time. The timer-based approach is actually preferred because the CPU can perform other tasks during the delay, and the delay time is not dependent on the exact number of instructions executed as it is in DLY50.

2.5.4 Assembler Listing

After a complete program or subprogram is written, it must be converted from mnemonics into binary machine code that the CPU can later execute. A separate computer system, such as an IBM PC, is used to perform this conversion to machine language. A computer program called an assembler is used. The assembler reads the mnemonic version of the program (also called the source version of the program) and produces a machine-code version of the program in a form that can be programmed into the memory of the MCU.

The assembler also produces a composite listing showing both the original source program (mnemonics) and the object code translation. This listing is used during the debug phase of a project and as part of the documentation for the software program. Figure 2-8 shows the listing which results from

```
               ****************************************************
               * Simple 68HC05 Program Example                    *
               * Read sw connected to bit-7 of port B; 1=closed   *
               * When sw. closes, light LED for about 1 Sec; LED  *
               * on when port C bit 6 = 0. Wait for sw release,   *
               * then repeat.  Debounce sw 50mS on & off          *
               ****************************************************
0001           PORTB   EQU   $01      Direct address of port B (sw)
0002           PORTC   EQU   $02      Direct address of port C (LED)
0005           DDRB    EQU   $05      Data direction control, port B
0006           DDRC    EQU   $06      Data direction control, port C
009f           TEMP1   EQU   $9F      One byte temp storage location

00a0                   ORG   $A0      Program will start at $00A0
               * $00A0 is in '705C8 RAM

00a0 a6 ff     INIT    LDA   #$FF     Begin initialization
00a2 b7 06             STA   DDRC     Set port C to act as outputs
               * Port B is configured as inputs by default from reset.
00a4 a6 e0             LDA   #$E0     Red & green LEDs and beeper off
00a6 b7 02             STA   PORTC    Turn off red LED
               * Some pins of port C (of my board) happen to be connected
               * to devices which don't apply to this example program.
               * The $E0 pattern turns off my stuff & turns off red LED

00a8 b6 01     TOP     LDA   PORTB    Read sw at MSB of Port B
00aa 2a fc             BPL   TOP      Loop till MSB=1 (Neg trick)
00ac cd 00 c3          JSR   DLY50    Delay about 50 mS to debounce
00af 1d 02             BCLR  6,PORTC  Turn on LED (bit-6 to zero)
00b1 a6 14             LDA   #20      Decimal 20 assembles to $14
00b3 cd 00 c3  DLYLP   JSR   DLY50    Delay 50 mS
00b6 4a                DECA           Loop counter for 20 loops
00b7 26 fa             BNE   DLYLP    20 times (20-19,19-18,.1-0)
00b9 1c 02             BSET  6,PORTC  Turn LED back off
00bb 0e 01 fd  OFFLP   BRSET 7,PORTB,OFFLP  Loop here till sw off
00be cd 00 c3          JSR   DLY50    Debounce release
00c1 20 e5             BRA   TOP      Look for next sw closure

               ***
               * DLY50 - Subroutine to delay ~50mS
               * Saves original accumulator value
               * but X will always be zero on return
               ***
00c3 b7 9f     DLY50   STA   TEMP1    Save accumulator in RAM
00c5 a6 20             LDA   #32      Do outer loop 32 times
00c7 5f        OUTLP   CLRX           X used as inner loop count
00c8 5a        INNRLP  DECX           0-FF, FF-FE,...1-0 256 loops
00c9 26 fd             BNE   INNRLP   6cyc*256*1µS/cyc = 1.536mS
00cb 4a                DECA           32-31, 31-30,...1-0
00cc 26 f9             BNE   OUTLP    1545cyc*32*1µS/cyc=49.440mS
00ce b6 9f             LDA   TEMP1    Recover saved Accumulator val
00d0 81                RTS            ** Return **
```

Figure 2-8. Assembler Listing

assembling the example program. Comments were added before the program was assembled.

Section 4 should be thoroughly studied before attempting to run any of the sample programs in this guide. Some of the sample programs were developed on another member of the M68HC05 Family which has a slightly different memory map than the MC68HC705C8. Minor modifications may be necessary to successfully run these programs on the MC68HC705C8.

Refer to Figure 2-9 for the following discussion. This figure shows some lines of the listing with reference numbers indicating the various parts of the line. The first line is an example of an assembler directive line. This line is not really part of the program; rather, it provides information to the assembler so that the real program can be converted properly into binary machine code.

```
0001           PORTB   EQU   $01     Direct address of port B (sw)

00a0                   ORG   $A0     Program will start at $00A0

00a8 b6 01     TOP     LDA   PORTB   Read sw at MSB of Port B
---- --------- ------- ----  ------- ------------------------------
[1]      [2]     [3]    [4]    [5]   [6]->
```

Figure 2-9. Explanation of Assembler Listing

EQU, short for equate, is used to give a specific memory location or binary number a name which can then be used in other program instructions. In this case, the EQU directive is being used to assign the name PORTB to the value $01, which is the address of port B in the MC68HC705C8. It is easier for a programmer to remember the mnemonic name PORTB rather than the annonymous numeric value $01. When the assembler encounters one of these names, the name is automatically converted to its corresponding binary value in much the same way that instruction mnemonics are converted into binary instruction codes.

The second line shown in Figure 2-9 is another assembler directive. The mnemonic ORG, which is short for originate, tells the assembler where the program will start (the address of the start of the first instruction following the ORG directive line). ORG directives may be used more than once in a program to tell the assembler to put different parts of the program in specific places in memory. Refer to the memory map of the MCU to select an appropriate memory location where a program should start.

In this assembler listing, the first two fields, [1] and [2], are generated by the assembler, and the last four fields, [3], [4], [5], and [6], are the original source program written by the programmer. Field [3] is a label (TOP) which can be referred to in other instructions. In our example program, the last instruction was "BRA TOP", which simply means the CPU will continue execution with the instruction that is labeled "TOP".

When the programmer is writing a program, the addresses where instructions will be located are not typically known. Worse yet, in branch instructions, rather than using the address of a destination, the CPU uses an offset (difference) between the current PC value and the destination address. Fortunately, the programmer does not have to worry about these problems because the assembler takes care of these details through a system of labels. This system of labels is a convenient way for the programmer to identify specific points in the program (without knowing their exact addresses); the assembler can later convert these mnemonic labels into specific memory addresses and even calculate offsets for branch instructions so that the CPU can use them.

Field [4] is the instruction field. The LDA mnemonic is short for load accumulator. Since there are six variations (different opcodes) of the load accumulator instruction, additional information is required before the assembler can choose the correct binary opcode for the CPU to use during execution of the program. Field [5] is the operand field, providing information about the specific memory location or value to be operated on by the instruction. The assembler uses both the instruction mnemonic and the operand specified in the source program to determine the specific opcode for the instruction.

The different ways of specifying the value to be operated on are called addressing modes (a more complete discussion of addressing modes is presented later). The syntax of the operand field is slightly different for each addressing mode so the assembler can determine the correct intended addressing mode from the syntax of the operand. In this case, the operand [5] is PORTB, which the assembler automatically converts to $01 (recall the EQU directive). The assembler interprets $01 as a direct addressing mode address between $0000 and $00FF, thus selecting the opcode $B6, which is the direct addressing mode variation of the LDA instruction. If PORTB had been preceded by a # symbol, that syntax would have been interpreted by the assembler as an immediate addressing mode value, and the opcode $A6 would have been chosen instead of $B6.

Field [6] is called the comment field and is not used by the assembler to translate the program into machine code. Rather, the comment field is used by the programmer to document the program. Although the CPU does not use this information during program execution, a good programmer knows that it is one of the most important parts of a good program. The comment [6] for this line of the program says "read sw at MSB of port B." This comment tells someone who is reading the listing why port B is being read, which is essential for understanding how the program works. An entire line can be made into a comment line by using an asterisk (*) as the first character in the line. In addition to good comments in the listing, it is also important to document programs with a flowchart or other detailed information explaining the overall flow and operation of the program.

2.5.5 CPU View of a Program

Figure 2-10, a memory map of the MC68HC705C8, shows how the example program fits in the memory of the MCU. This figure is the same as Figure 2-4 except that a different portion of the memory space has been expanded to show the contents of all locations in the program. Figure 2-10 shows that the CPU sees the example program as a linear sequence of binary codes, including instructions and operands in successive memory locations. The CPU begins this program with its program counter (PC) pointing at the first byte in the program. Each instruction opcode tells the CPU how many (if any) and what type of operands go with that instruction. In this way, the CPU can remain aligned to instruction boundaries even though the mixture of opcodes and operands looks confusing to us.

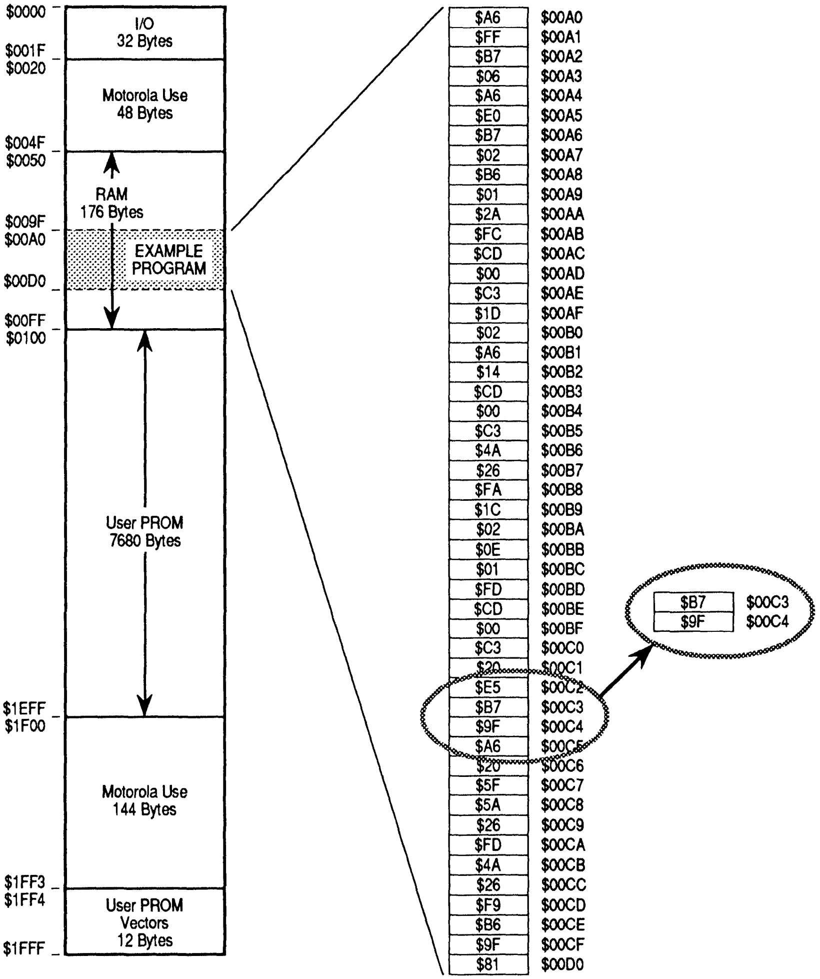
Most application programs would be located in ROM, EPROM, or OTPROM. This example program is loaded into an area of RAM to avoid having to program (and later erase) the EPROM. There is no special requirement that instruction must be in a ROM-type memory to execute. As far as the CPU is concerned, any program is just a series of binary bit patterns which are sequentially processed.

Carefully study the program listing in Figure 2-8 and the memory map of Figure 2-10. Find the first instruction of the DLY50 subroutine in Figure 2-8 and then find the same two bytes in Figure 2-10.

You should have found the following line from near the bottom of Figure 2-8.

```
00c3 b7 9f     DLY50   STA    TEMP1     Save accumulator in RAM
```

The outlined section of memory in Figure 2-10 is the area you should have identified.



$A6	$00A0
$FF	$00A1
$B7	$00A2
$06	$00A3
$A6	$00A4
$E0	$00A5
$B7	$00A6
$02	$00A7
$B6	$00A8
$01	$00A9
$2A	$00AA
$FC	$00AB
$CD	$00AC
$00	$00AD
$C3	$00AE
$1D	$00AF
$02	$00B0
$A6	$00B1
$14	$00B2
$CD	$00B3
$00	$00B4
$C3	$00B5
$4A	$00B6
$26	$00B7
$FA	$00B8
$1C	$00B9
$02	$00BA
$0E	$00BB
$01	$00BC
$FD	$00BD
$CD	$00BE
$00	$00BF
$C3	$00C0
$20	$00C1
$E5	$00C2
$B7	$00C3
$9F	$00C4
$A6	$00C5
$20	$00C6
$5F	$00C7
$5A	$00C8
$26	$00C9
$FD	$00CA
$4A	$00CB
$26	$00CC
$F9	$00CD
$B6	$00CE
$9F	$00CF
$81	$00D0

Figure 2-10. Memory Map of Example Program

2.6 CPU OPERATION

This section will first discuss the detailed operation of CPU instructions and then explain how the CPU would execute the example program. The detailed descriptions of typical CPU instructions are intended to make you think like a CPU. We can then go through the example program using a teaching technique called 'playing computer' in which you pretend you are the CPU interpreting and executing the instructions in a program.

2.6.1 Detailed Operation of CPU Instructions

Before seeing how the CPU would execute the example program, it would help to know (in detail) how the CPU breaks down instructions into fundamental operations and performs these tiny steps to accomplish a desired instruction. As we will see, many small steps execute very quickly and very accurately within each instruction, but none of the small steps is very complicated.

The logic circuitry inside the CPU would seem straightforward to a design engineer accustomed to working with TTL logic or even relay logic. What sets the MCU and its CPU apart from these other forms of digital logic is the packing density. Very large scale integration (VLSI) techniques have made it possible to fit the equivalent of thousands of TTL integrated circuits on a single silicon die. By arranging these logic gates to form a CPU, you get a general-purpose instruction executer capable of acting as a universal logic element. By placing different combinations of instructions in the device, it can perform virtually any definable function.

A typical instruction takes two to five cycles of the internal processor clock. Although it is not normally important to know exactly what happens during each of these execution cycles, it can help to go through a few instructions in detail to understand how the CPU works internally.

2.6.1.1 STORE ACCUMULATOR (DIRECT ADDRESSING MODE). Look up the STA instruction in Appendix A. In the table at the bottom of the page, we see that $B7 is the direct addressing mode version of the store accumulator instruction. We also see that the instruction requires two bytes, one to specify the opcode ($B7) and the second to specify the direct address where the accumulator will be stored. (The two bytes are shown as "B7 dd" in the machine code column of the table.)

We will be discussing the addressing modes in more detail later, but the following brief description will help in understanding how the CPU executes this instruction. In direct addressing modes, the CPU assumes the address is in the range of $0000 through $00FF; thus, there is no need to include the upper byte of address of the operand in the instruction (since it is always $00).

The table at the bottom of the STA page shows that the direct addressing version of the STA instruction takes four CPU cycles to execute. During the first cycle of this STA instruction, the CPU reads the opcode $B7, which identifies the instruction as the direct addressing version of the STA instruction and advances the PC to the next memory location.

During the second cycle, the CPU places the value from the PC on the internal address bus and reads the low-order byte of the direct address ($02 for example). The CPU uses the third cycle of this STA instruction to internally construct the full address where the accumulator is to be stored, and also advances the PC so it points to the next address in memory (the address of the opcode of the next instruction).

In this example, the CPU appends the assumed value $00 (because of direct addressing mode) to the $02 that was read during the second cycle of the instruction to arrive at the complete address $0002. During the fourth cycle of this instruction, the CPU places this constructed address ($0002) on the internal address bus, places the accumulator value on the internal data bus, and asserts the write signal. That is, the CPU writes the contents of the accumulator to $0002 during the fourth cycle of the STA instruction.

This explanation left out certain details, such as setting the condition code flags, but it gives an idea of what occurs within the CPU during the execution of a single instruction.

2.6.1.2 LOAD ACCUMULATOR (IMMEDIATE ADDRESSING MODE). Next, look up the LDA instruction in the instruction set. The immediate addressing mode version of this instruction appears as "A6 ii" in the machine code column of the table at the bottom of the page. This version of the instruction takes two internal processor clock cycles to execute.

The $A6 opcode tells the CPU to get the byte of data that immediately follows the opcode and put this value in the accumulator. During the first cycle of this instruction, the CPU reads the opcode $A6 and advances the PC to point to the next location in memory (the address of the immediate operand ii). During the second cycle of the instruction, the CPU reads the contents of the byte following the opcode into the accumulator and advances the PC to point at the next location in memory (i.e., the opcode byte of the next instruction).

While the accumulator was being loaded, the N and Z bits in the accumulator were set or cleared according to the data that was loaded into the accumulator. The boolean logic formulae for these bits appears near the middle of the instruction set page. The Z bit will be set if the value loaded into the accumulator was $00; otherwise, the Z bit will be cleared. The N bit will be set if the most significant bit of the value loaded was a logic one; otherwise, N will be cleared.

The N (negative) condition code bit may be used to detect the sign of a twos-complement number. In twos-complement numbers, the most significant bit is used as a sign bit, one indicates a negative value, and zero indicates a positive value. The N bit may also be used as a simple indication of the state of the most significant bit of a binary value.

2.6.1.3 CONDITIONAL BRANCH. Branch instructions allow the CPU to select one of two program flow paths, depending upon the state of a particular bit in memory or various condition code bits. If the condition checked by the branch instruction is true, program flow proceeds to a specified location in memory. If the condition checked by the branch is not true, the CPU proceeds to the instruction following the branch instruction. Decision blocks in a flowchart correspond to conditional branch instructions in the program.

Most branch instructions contain two bytes, one for the opcode and one for a relative offset byte. Branch on bit clear (BRCLR) and branch on bit set (BRSET) instructions require three bytes: the opcode, a one-byte direct address (to specify the memory location to be tested), and the relative offset byte.

The relative offset byte is interpreted by the CPU as a twos-complement signed value. If the branch condition checked is true, this signed offset is added to the PC, and the CPU reads its next instruction from this calculated new address. If the branch condition is not true, the CPU just continues to the next instruction after the branch instruction.

The following excerpt from Figure 2-8 demonstrates a useful way to use a conditional branch based on the N condition code bit that is sometimes overlooked.

```
00a8 b6 01      TOP     LDA    PORTB     Read sw at MSB of Port B
00aa 2a fc              BPL    TOP       Loop till MSB=1 (Neg trick)
00ac cd 00 c3           JSR    DLY50     Delay about 50 mS to debounce
```

The first line means "load accumulator with the value at I/O port B of the MCU." The most significant bit of this port is connected to a normally opened switch and a pulldown resistor. When the switch is pressed (closed), a logic one is applied to the port pin. If the LDA PORTB instruction is executed when the switch is opened, the N condition code bit will be cleared. Conversely, if the LDA PORTB instruction is executed when the switch is closed, the N condition code bit will be set.

The second line in the listing (BPL TOP) is read "branch if plus to TOP." In response to this instruction, the CPU either branches back to the first line of this program or falls to the third line of the program, depending on the condition of the N condition code bit. If the N condition code bit is clear, the CPU branches to the first line of the program. This corresponds to the CPU interpreting the value previously read from port B as a positive value; hence, the instruction name "branch if plus."

Tricks such as that just described are not the only way to read and respond to I/O conditions. The following two lines of code would accomplish the same effect as the three lines which used the N-bit trick.

```
00a8 0f 01 fd      TOP  BRCLR  7,PORTB,TOP    Loop till sw closed
00ab cd 00 c3           JSR    DLY50          Delay about 50 mS to debounce
```

The first line of this sequence is read "branch to TOP if bit 7 of port B is clear." In this particular case, the second sequence is better than the first sequence for several reasons. The second squence is more straightforward (less chance for confusion), it takes one less byte of machine code, and it executes one cycle faster than the three-line sequence. However, in some cases the operand (PORTB) is needed in the accumulator for some other reason; thus, the first instruction sequence based on the N-bit trick becomes the slightly better choice. From a practical point of view, the differences between these two approaches is very small, and either would work well in an application.

2.6.1.4 SUBROUTINE CALLS AND RETURNS. The jump-to-subroutine (JSR) and branch-to-subroutine (BSR) instructions automate the process of leaving the normal linear flow of a program to go off and execute a set of instructions and then return to where the normal flow left off. The set of instructions outside the normal program flow is called a subroutine. A JSR or BSR instruction is used to go from the running program to the subroutine and a return-from-subroutine (RTS) instruction is used to return to the program from which the subroutine was called.

The following figure shows lines of an assembler listing which will be used to demonstrate how the CPU executes a subroutine call. Assume that the stack pointer (SP) points to address $00FF when the CPU encounters the JSR instruction at location $0102.

```
0100 a6 02       TOP     LDA    #$02      Load an immediate value
0102 cd 02 00            JSR    SUBBY     Go do a subroutine
0105 b7 02               STA    $02       Store accumulator to port C
  "    "                  "      "          "
  " .  "                  "      "          "
  "    "                  "      "          "
0200 4a          SUBBY   DECA             Decrement accumulator
0201 26 fd               BNE    SUBBY     Loop till accumulator=0
0203 81                  RTS              ** Return from subroutine **
```

Refer to Figure 2-11 during the following discussion. We will begin the explanation with the CPU executing the instruction "LDA #$02" at address $0100. The left side of the figure shows the normal program flow composed of TOP LDA #$20, JSR SUBBY, and STA $02 (in that order) in consecutive memory locations. The right half of the figure shows subroutine instructions SUBBY DECA, BNE SUBBY, and RTS.

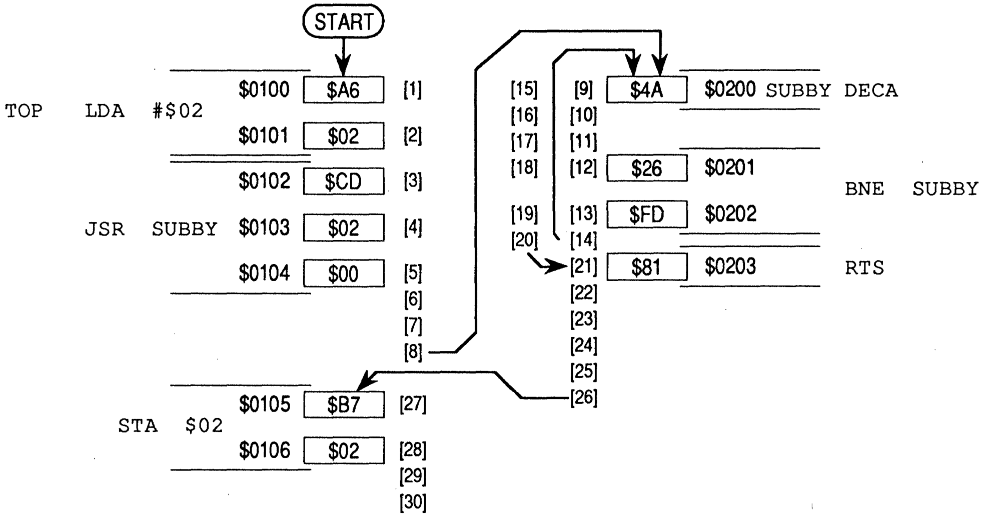


Figure 2-11. Subroutine Call Sequence

Each number in square brackets indicates a cycle of the internal processor clock. The cycle numbers will be used as references in the following explanation of this figure.

[1] CPU reads $A6 opcode from location $0100 (LDA immediate).

[2] CPU reads immediate data $02 from location $0101 into the accumulator.

[3] CPU reads $CD opcode from location $0102 (JSR extended).

[4] CPU reads high-order extended address $02 from $0103.

[5] CPU reads low-order extended address $00 from $0104.

[6] CPU builds full address of subroutine ($0200).

[7] CPU writes $05 to $00FF and decrements SP to $00FE. Another way to say this is "push low-order half of return address on stack."

[8] CPU writes $01 to $00FE and decrements SP to $00FD. Another way to say this is "push high-order half of return address on stack." The return address that was saved on the stack is $0105, which is the address of the instruction that follows the JSR instruction.

[9] CPU reads $4A opcode from location $0200. This is the first instruction of the called subroutine.

[10] [11] The DECA instruction takes three cycles ([9], [10], and [11]).

[12] CPU reads BNE opcode ($26) from location $0201.

[13] CPU reads relative offset ($FD) from $0202.

[14] During the LDA #$02 instruction at [1], the accumulator was loaded with the value 2; during the DECA instruction at [9], the accumulator was decremented to 1 (which is not equal to zero). Thus, at [14] the branch condition was true, and the twos-complement offset ($FD or −3) was added to the internal PC (which was $0203 at the time) to get the value $0200.

[15] through [19] are a repeat of cycles [9] through [13] except that when the DECA instruction at [15] was executed this time, the accumulator went from $01 to $00.

[20] Since the accumulator is now "equal to zero," the BNE [19] branch condition is not true, and the branch will not be taken.

[21] CPU reads the RTS opcode ($81) from $0203.

[22] through [26] The RTS takes six cycles. During the last five cycles of this instruction, the SP is incremented to $00FE, the high-order return address ($01) is read from the stack ($00FE), the SP is incremented again to $00FF, the low-order return address ($05) is read from the stack ($00FF), and the PC is loaded with this recovered return address ($0105).

[27] CPU reads the STA direct opcode ($B7) from location $0105.

[28] CPU reads the low-order direct address ($02) from location $0106.

[29] [30] The STA direct instruction takes a total of four cycles. During these last two cycles of the instruction, the CPU constructs the complete address where the accumulator will be stored by appending $00 (assumed value for the high-order half of the address due to direct addressing mode) to the $02 read during [28]. The accumulator ($00 at this time) is then stored to this constructed address ($0002).

2.6.2 Playing Computer

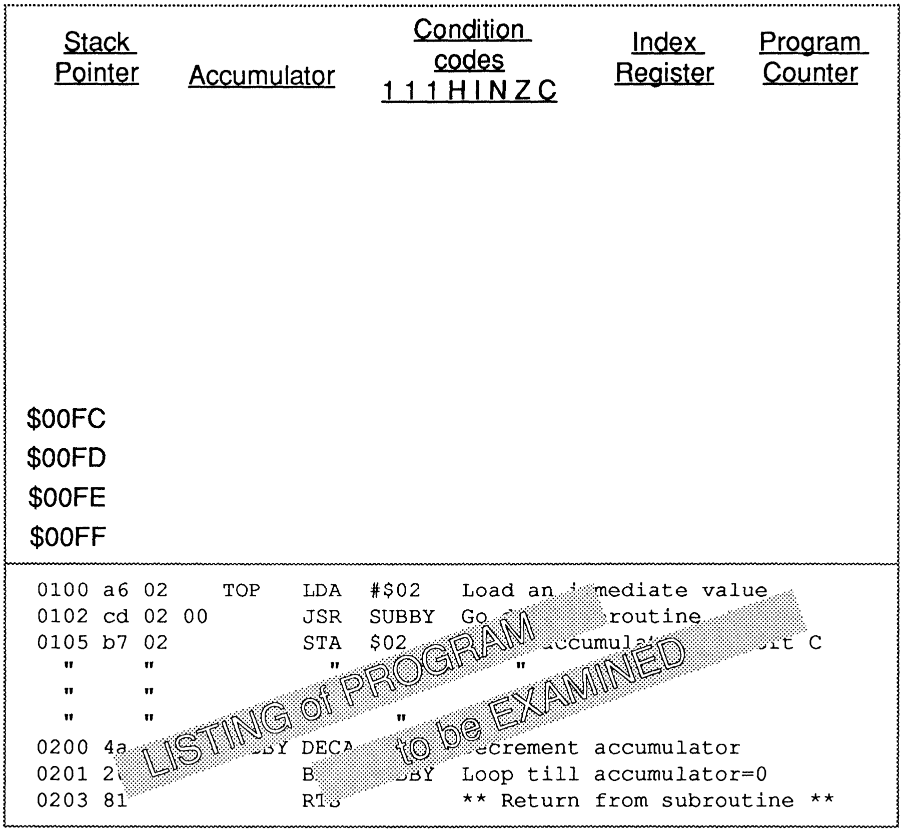
Playing computer is a learning exercise where you pretend to be a CPU that is executing a program. Programmers often mentally check programs by playing computer as they read through a software routine. While playing computer, it is not necessary to break instructions down to individual processor cycles. Instead, instructions are treated as a single complete operation rather than several detailed steps.

The following paragraphs demonstrate the process of playing computer by going through the subroutine-call exercise of Figure 2-11. The playing-computer approach to analyzing this sequence is much less detailed than the cycle-by-cycle analysis done earlier on Figure 2-11, but it accomplishes the same basic goal — i.e., it shows what happens as the CPU executes the sequence. After seeing how to do this exercise, you should attempt the same thing with a larger program such as the example of Figure 2-10.

You begin the process by preparing a worksheet like that shown in Figure 2-12. This sheet includes the mnemonic program and the machine code that it assembles to. (You could alternately choose to use a listing positioned next to the worksheet.) The worksheet also includes the CPU register names across the top of the sheet with ample of room below to write new values as the registers change in the course of the program.

On this worksheet, there is an area for keeping track of the stack. After you become comfortable with how the stack works, you would probably leave this section off, but it will be instructive to leave it here for now.

As a value is saved on the stack, you will cross out any prior value and write the new value to its right in a horizontal row. You must also update (decrement) the SP value by crossing out any prior value and writing the new value beneath it under the SP heading at the top of the worksheet. As a value is recovered from the stack, you would update (increment) the value of SP by crossing out the old value and writing the new value below it. You would then read the value from the location now pointed to by the SP and put it wherever it belongs in the CPU (e.g., in the upper or lower half of the PC).

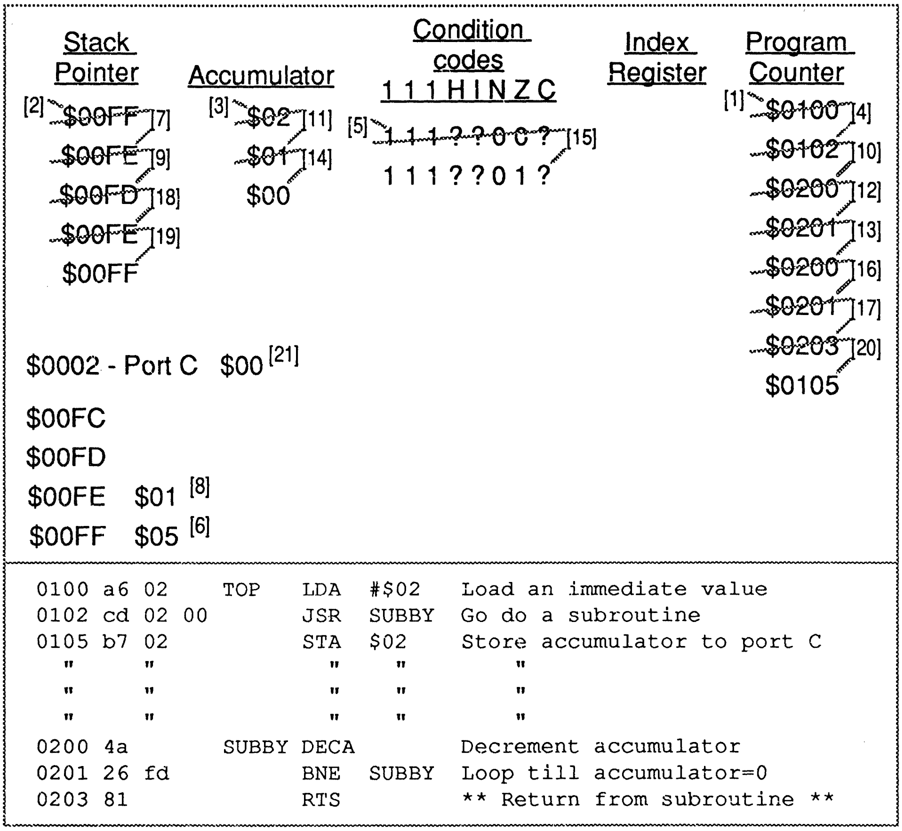


```
0100 a6 02     TOP    LDA   #$02    Load an    mediate value
0102 cd 02 00         JSR   SUBBY   Go           routine
0105 b7 02            STA   $02            ccumula          it C
  "     "               "              "
  "     "
  "     "                        "
0200 4a              BY DECA            ecrement accumulator
0201 2                B          BY   Loop till accumulator=0
0203 81               RT              ** Return from subroutine **
```

Figure 2-12. Playing Computer Worksheet

Figure 2-13 shows how the worksheet will look after working through the whole JSR sequence. Follow the numbers in square brackets as the process is explained. During the process, many values were written and later crossed out; a line has been drawn from the square bracket to either the value or the crossed out mark to show which item the reference number applies to.

Beginning the sequence, the PC should be pointing to $0100 [1], and the SP should be pointing to $00FF [2] (due to an earlier assumption). The CPU reads and executes the LDA #$02 instruction (load accumulator with the immediate value $02); thus, you write $02 in the accumulator column [3] and replace the PC value [4] with $0102, which is the address of the next instruction. The



Stack Pointer	Accumulator	Condition codes 1 1 1 H I N Z C	Index Register	Program Counter
[2] ~~$00FF~~ [7]	[3] ~~$02~~ [11]	[5] ~~1 1 1 ? ? 0 0 ?~~ [15]		[1] ~~$0100~~ [4]
~~$00FE~~ [9]	~~$01~~ [14]	1 1 1 ? ? 0 1 ?		~~$0102~~ [10]
~~$00FD~~ [18]	$00			~~$0200~~ [12]
~~$00FE~~ [19]				~~$0201~~ [13]
$00FF				~~$0200~~ [16]
				~~$0201~~ [17]
				~~$0203~~ [20]
				$0105

$0002 - Port C $00 [21]

$00FC
$00FD
$00FE $01 [8]
$00FF $05 [6]

```
0100 a6 02      TOP   LDA  #$02   Load an immediate value
0102 cd 02 00         JSR  SUBBY  Go do a subroutine
0105 b7 02            STA  $02    Store accumulator to port C
  "    "                "    "        "
  "    "                "    "        "
  "    "                "    "        "
0200 4a         SUBBY DECA        Decrement accumulator
0201 26 fd            BNE  SUBBY  Loop till accumulator=0
0203 81               RTS         ** Return from subroutine **
```

Figure 2-13. Completed Worksheet

load accumulator instruction affects the N and Z CCR bits. Since the value loaded was $02, the Z bit would be cleared, and the N bit would be cleared [5]. This information can be found in Appendix A. Since the other bits in the CCR are not affected by the LDA instruction, we have no way of knowing what they should be at this time, so we put question marks in the unknown positions for now [5].

Next, the CPU reads the JSR SUBBY instruction. Temporarily remember the value $0105, which is the address where the CPU should come back to after executing the called subroutine. The CPU saves the low-order half of the return address on the stack; thus, you write $05 [6] at the location pointed to by the SP ($00FF) and decrement the SP [7] to $00FE. The CPU then saves the high-order half of the return address on the stack; you write $01 [8] to $00FE and again decrement the SP [9] (this time to $00FD). To finish the JSR instruction, you load the PC with $0200 [10], which is the address of the called subroutine.

The CPU fetches the next instruction. Since the PC is $0200, the CPU executes the DECA instruction, the first instruction in the subroutine. You cross out the $02 in the accumulator column and write the new value $01 [11]. You also change the PC to $0201 [12]. Because the DECA instruction changed the accumulator from $02 to $01 (which is not zero or negative), the Z bit and N bit remain clear. Since N and Z were already cleared at [5], you can leave them alone on the worksheet.

The CPU now executes the BNE SUBBY instruction. Since the Z bit is clear, the branch condition is met, and the CPU will take the branch. Cross out the $0201 under PC and write $0200 [13].

The CPU again executes the DECA instruction. The accumulator is now changed from $01 to $00 [14] (which is zero and not negative); thus, the Z bit is set, and the N bit remains clear [15]. The PC advances to the next instruction [16].

The CPU now executes the BNE SUBBY instruction, but this time the branch condition is not true (Z is set now), so the branch will not be taken. The CPU simply falls to the next instruction (the RTS at $0203). Update the PC to $0203 [17].

The RTS instruction causes the CPU to recover the previously stacked PC. Pull the high-order half of the PC from the stack by incrementing the SP to $00FE [18] and by reading $01 from location $00FE. Next, pull the low-order half of the address from the stack by incrementing SP to $00FF [19] and by reading $05 from $00FF. The address recovered from the stack replaces the value in the PC [20].

The CPU now reads the STA $02 instruction from location $0105. Program flow has returned to the main program sequence where it left off when the subroutine was called. The STA (direct addressing mode) instruction writes the accumulator value to the direct address $02 ($0002), which is port C on the MC68HC705C8. We can see from the worksheet that the current value in the accumulator is $00; therefore, all eight pins of port C would be driven low (provided they are configured as outputs at this time). Since the original worksheet did not have a place marked for recording the value of port C, you would make a place and write $00 there [21].

For a larger program, the worksheet would have many more crossed out values by the time you are done. Playing computer on a worksheet like this is a good learning exercise, but, as a programmer gains experience, the process would be simplified.

One of the first simplifications would be to quit keeping track of the PC because you learn to trust the CPU to take care of this for you. Another simplification of the worksheet is to stop keeping track of the condition codes. When a branch instruction which depends on a condition code bit is encountered, you can mentally work backwards to decide whether or not the branch should be taken.

Next, the storage of values on the stack would be skipped, although it is still a good idea to keep track of the SP value because it is fairly common to have programming errors resulting from incorrect values in the SP. A fundamental operating principle of the stack is that over a period of time, the same number of items must be removed from the stack as were put on the stack. Just as left parentheses must be matched with right parentheses in a mathematical formula, JSRs and BSRs must be matched one for one to subsequent RTSs in a program. Errors which cause this rule to be broken will appear as erroneous SP values while playing computer.

Even an experienced programmer will play computer occasionally to solve some difficult problem. The procedure the experienced programmer would use is much less formal than what was explained here, but it still amounts to placing yourself in the role of the CPU and working out what happens as the program is executed.

2.7 ON-CHIP PERIPHERALS

A peripheral is a block of circuitry which performs some useful function under control of the CPU. One example of a peripheral is a universal asynchronous receiver/transmitter (UART), which acts as an interface between a computer and an asynchronous serial communication link. The most common example of such a communication link is the RS-232 or RS-422 serial port on a computer. This standard is so universal that almost every personal and mainframe computer made anywhere in the world has at least one such port.

Before the MCU was developed, a computer designer had to use a separate UART integrated circuit to include this serial interface function in a computer. Often a number of other miscellaneous logic gates were also needed to interface the UART to the CPU buses.

Since the level of integration allows thousands of logic gates to be included in a single MCU integrated circuit, it is practical to put several peripherals, including this UART function, on the same chip along with the CPU and memories. The on-chip serial communications interface (SCI) in the MC68HC705C8 is a UART-type peripheral.

It is important for the MCU manufacturer to select peripheral functions that will be useful to many potential users for inclusion on the MCU chip. This pressure to make on-chip peripherals satisfy the requirements of as many customers as possible causes the need for user-selectable options to modify the operation of the on-chip peripherals.

The MC68HC705C8 has control registers, which allow a user to select which parallel I/O pins will be inputs and which will be outputs. Although any one application is likely to need only one specific mixture of inputs and outputs, twenty different applications are likely to need a dozen collective mixtures. The ability to specify the direction of each I/O pin at the time of use makes this MCU ideal for many different applications.

Control registers are controlled by the CPU in essentially the same way as a digital output port. You could think of control/status registers as internal I/O registers connected to internal logic rather than to MCU pins. To change the voltage level at an output pin, the CPU writes a digital value to the address of the output port register. The level (0 or 1) in each bit of the output port register controls the voltage level on a corresponding MCU pin. In the case of a control register, the state of a bit in the control register determines the logic level of an internal control signal rather than on a pin.

In Section 3 of this applications guide, you will find more complete descriptions of the on-chip peripherals in the MC68HC705C8.

2.7.1 Serial Communications Interface (SCI)

The SCI system on the MC68HC705C8 is a UART-type asynchronous serial communications interface. The most common use of this peripheral is to implement an RS-232 interface to a host computer system (such as a personal computer). The SCI system can be used to communicate over relatively long distances.

In normal applications, the CPU simply writes data to a parallel data register to send a formatted serial character. The SCI peripheral system takes care of all the details of transforming the data into the proper serial format, including the addition of start and stop bits required to meet standards. The transmitter even allows up to two characters to be queued up for transmission, thus allowing the CPU more time to prepare additional characters.

The receiver portion of the SCI automatically detects the start of a character and intelligently samples the incoming serial data to assure correct reception, even in noisy applications. All activity related to receiving serial data and converting it to parallel data is performed within the SCI peripheral logic with no intervention of the CPU. After a character is received, the CPU simply reads a data byte from a receive data register.

A number of options are offered to allow various data rates (baud rates), alternate character formats, and an automatic standby/wakeup feature. You can choose between software polling or interrupts for detection of SCI status.

2.7.2 Serial Peripheral Interface (SPI)

The SPI system on the MC68HC705C8 is separate from the SCI system and is used primarily for communications with standard peripheral logic chips on the same circuit board as the MCU. A few examples of the chips that can use SPI are serial-to-parallel and parallel-to-serial shift registers, A/D peripherals, LCD peripherals, and many others.

The SPI system works like a distributed 16-bit shift register in which half the shifter is in the MCU (SPI), and the other half is in the peripheral. When the MCU initiates a transfer, this distributed shifter is rotated eight bit positions so that the data in the master MCU is effectively exchanged with the data in the peripheral slave. In some cases, the loop is incomplete, and data is transferred only from the MCU to the peripheral or from the peripheral to the MCU.

An SPI system typically consists of a master MCU and one or more slave peripherals. Other configurations such as two MCUs or multiple master systems are possible but less common.

The SPI system includes options to select shift rate, master or slave mode, clock polarity, and phase to allow compatibility with most synchronous serial peripheral devices from many manufacturers.

2.7.3 16-Bit Timer System

The MC68HC705C8 MCU includes a 16-bit timer system used to measure time and to produce signals of specific period or frequency. This system is based on a free-running 16-bit counter, a 16-bit output-compare register, and a 16-bit input-capture register.

The CPU controls the timing of output signals through the output-compare mechanism. To schedule an output change to occur at a specific time (a specific count of the free-running counter), a 16-bit value corresponding to the desired time is written to the output-compare register. When the free-running counter matches the value in the output-compare register, the planned output change occurs.

The CPU detects the time of an event or measures the period of an input signal with the input-capture mechanism. The CPU can select either positive or negative edges detected on an MCU pin to trigger the input-capture mechanism. When the selected edge occurs, the current value in the free-running counter (which corresponds to the time the edge occurred), is captured by (transferred to) the input-capture register. The CPU can later read the value in the input-capture register and determine the exact time when the edge occurred.

2.7.4 Memory Peripherals

Memory systems are also a form of peripheral. The uses for different types of memory were discussed earlier, but the logic required to support these memories was not discussed. ROM and RAM memories are very straightforward and require no support logic other than address-select logic to distinguish one location from another.

EPROM (erasable programmable ROM) and EEPROM (electrically erasable programmable ROM) memories require support logic for programming (and erasure in the case of EEPROM). The peripheral support logic in the MC68HC705C8 is like having a PROM programmer built into the MCU. A control register includes control bits to select between programming and normal modes and to enable the high-voltage programming supply.

2.7.5 Other On-Chip Peripherals

There are many other peripherals available on MCUs (see other members of the M68HC05 Family of MCUs). These other peripherals include analog-to-digital (A/D) converters, liquid crystal display drivers (LCD),and vacuum flourescent display drivers (VFD).

2

SECTION 3
MC68HC705C8 FUNCTIONAL DATA

The MC68HC705C8 microcontroller (MCU) is a member of the M68HC05 Family of low-cost, single-chip microcontrollers.

The HCMOS technology used on the MC68HC705C8 combines smaller size and higher speeds with the low power and high noise immunity of CMOS.

An additional advantage of CMOS is that circuitry is fully static. CMOS microcontrollers may be operated at any clock rate less than the guaranteed maximum. This feature may be used to conserve power since power consumption increases with higher clock frequencies. Static operation may also be advantageous during product development.

Two software-controlled power-saving modes, WAIT and STOP, are available to conserve additional power. These modes make the MC68HC705C8 especially attractive for automotive and battery-driven applications.

3.1 MCU DESCRIPTION

The hardware and software highlights of the MC68HC705C8 are as follows:

Hardware Features

- HCMOS Technology
- 8-Bit Architecture
- Power-Saving Stop, Wait, and Data Retention Modes
- 24 Bidirectional I/O Lines
- 7 Input-Only Lines
- 2 Timer I/O Pins
- 2.1 MHz Internal Operating Frequency, 5 Volts; 1.0 MHz, 3 Volts
- Internal 16-Bit Timer
- Serial Communications Interface (SCI) System
- Serial Peripheral Interface (SPI) System

- Ultraviolet (UV) light EPROM or One-Time Programmable ROM (OTPROM)
- Selectable Memory Configurations
- Computer Operating Properly (COP) Watchdog System
- Clock Monitor
- On-Chip Bootstrap Firmware for Programming
- Software-Programmable External Interrupt Sensitivity
- External Pin, Timer, SCI, and SPI Interrupts
- Master Reset and Power-On Reset
- Single 3- to 6-Volt Supply (2-Volt Data Retention Mode)
- On-Chip Oscillator
- 40-Pin Dual-in-Line Package or
- 44-Lead PLCC (Plastic Leaded Chip Carrier) Package

Software Features

- Upward Software Compatible with the M146805 CMOS Family
- Efficient Instruction Set
- Versatile Interrupt Handling
- True Bit Manipulation
- Addressing Modes with Indexed Addressing for Tables
- Memory-Mapped I/O
- Two Power-Saving Standby Modes

Figure 3-1 shows the MC68HC705C8 MCU block diagram.

The central processor unit (CPU) contains the 8-bit arithmetic logic unit, accumulator, index register, condition code register, stack pointer, program counter, and CPU control logic.

Major peripheral functions are provided on-chip. On-chip memory systems include bootstrap read-only memory (ROM), programmable ROM (EPROM or OTPROM), and random-access memory (RAM).

On-chip I/O devices include an asynchronous serial communications interface (SCI), a separate serial peripheral interface (SPI), and a 16-bit programmable timer system.

Self-monitoring circuitry is included on-chip to protect against system errors. A computer operating properly (COP) watchdog system protects against software failures. A clock monitor system generates a system reset if the clock is lost or runs too slow. An illegal opcode detection circuit provides a nonmaskable interrupt if an illegal opcode is detected.

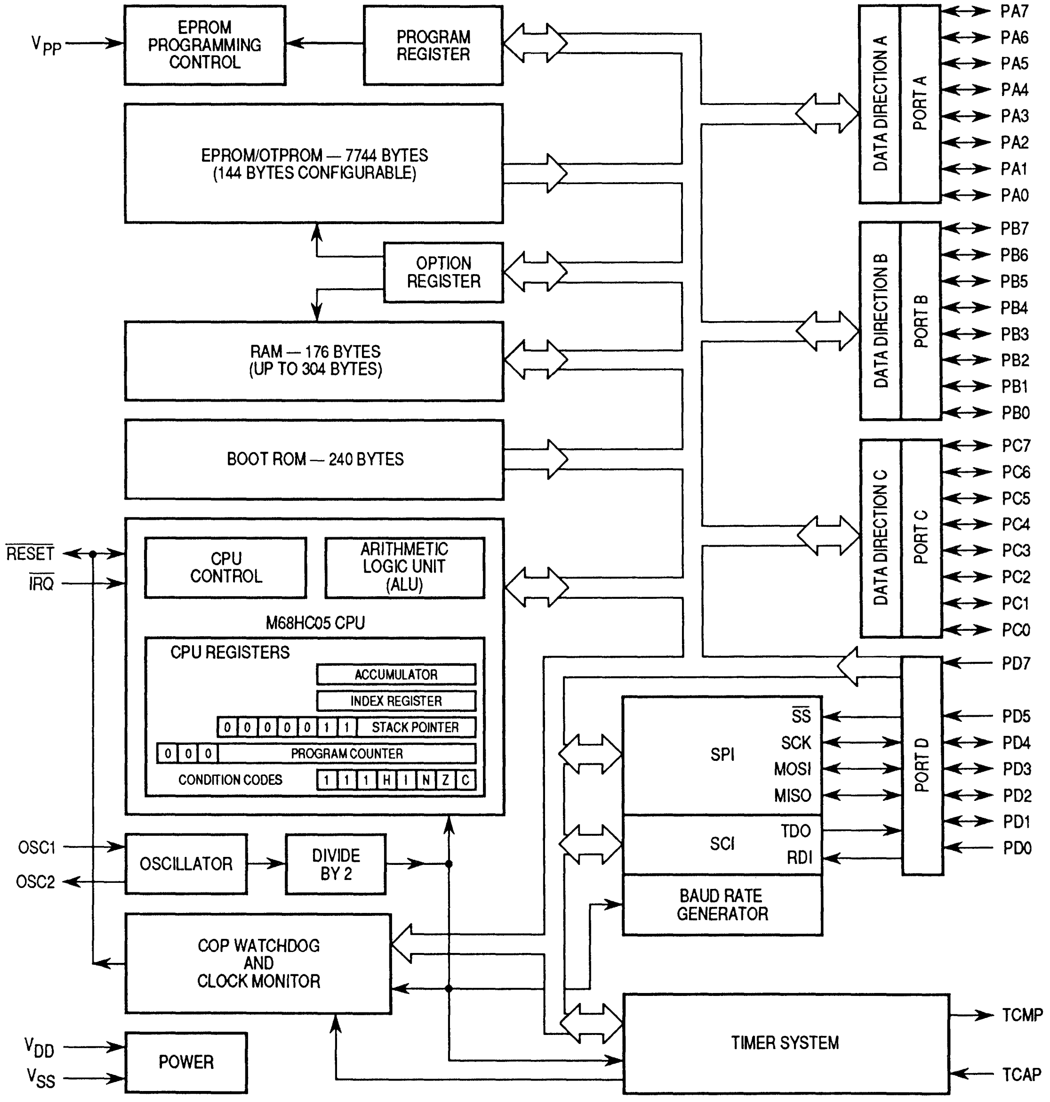


Figure 3-1. MC68HC705C8 Microcontroller Block Diagram

3.2 PINS AND CONNECTIONS

The following paragraphs discuss the MCU pin assignments, pin functions, and basic connections.

Because the MC68HC705C8 is a CMOS device, unused input pins must be terminated to avoid oscillation, noise, and added supply current. The preferred method of terminating pins that can be configured for input or output is with individual pullup or pulldown resistors for each unused pin.

Pin assignments are shown in Figure 3-2. Mechanical data and ordering information can be found in BR594/D, the *MC68HC705C8 Technical Summary*, available separately.

3.2.1 Pin Functions

V_{DD} and V_{SS}

Power is supplied to the MCU using these two pins. V_{DD} is power and V_{SS} is ground. The MCU can operate from a single 5-volt (nominal) power supply.

V_{PP}

The V_{PP} pin is used when programming the one-time programmable ROM (OTPROM) or EPROM. Programming voltage (14.75 Vdc) is applied to this pin when programming the PROM. Normally, this pin is connected to V_{DD}.

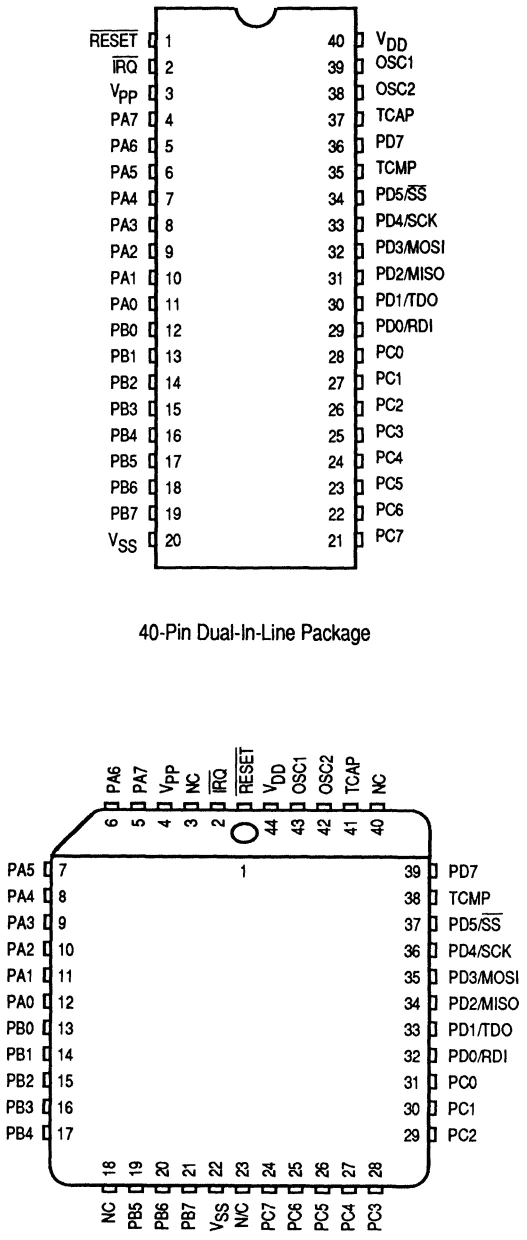
CAUTION

Do not connect V_{PP} pin to V_{SS} (GND). It will damage the MCU.

$\overline{\text{IRQ}}$ (Maskable Interrupt Request)

$\overline{\text{IRQ}}$ is a software programmable option which provides two different choices of interrupt triggering sensitivity. These options are 1) negative edge-sensitive triggering only, or 2) both negative edge-sensitive and level-sensitive triggering.

In the latter case, either a negative edge or a low level input to the $\overline{\text{IRQ}}$ pin will produce an interrupt. The MCU completes the current instruction before it responds to the interrupt request. When the $\overline{\text{IRQ}}$ pin goes low, a



40-Pin Dual-In-Line Package

44-Lead PLCC Package

Figure 3-2. Pin Assignments

small synchronization delay occurs, and a logic one is latched internally to signify an interrupt has been requested. When the MCU completes its current instruction, the interrupt latch is tested. If the interrupt latch contains a logic one and the interrupt mask bit (I bit) in the condition code register is clear, the MCU then begins the interrupt sequence.

If the option is selected to include level-sensitive triggering, then the $\overline{\text{IRQ}}$ input requires an external resistor to V_{DD} for "wired-OR" operation. See **3.4.7 Interrupts** for more detail concerning interrupts.

$\overline{\text{RESET}}$

The $\overline{\text{RESET}}$ pin is an active-low bidirectional control signal. As an input, the $\overline{\text{RESET}}$ pin initializes the MCU to a known startup state. As an open-drain output, the $\overline{\text{RESET}}$ pin indicates an internal MCU failure detected by the computer operating properly (COP) watchdog timer or clock monitor circuitry.

This $\overline{\text{RESET}}$ pin is significantly different from the RESET signal used on other Motorola M68HC05 Family devices. Refer to **3.4.4 Resets** and **3.4.7 Interrupts** before designing circuitry to generate or monitor the RESET signal.

TCAP

The TCAP pin provides the input to the input-capture feature for the on-chip programmable timer system. Refer to input-capture register in **3.8 PROGRAMMABLE TIMER**.

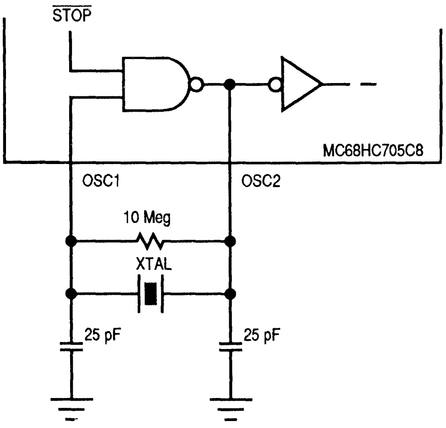
TCMP

The TCMP pin provides an output for the output-compare feature of the on-chip timer system. Refer to output-compare register in **3.8 PROGRAMMABLE TIMER**.

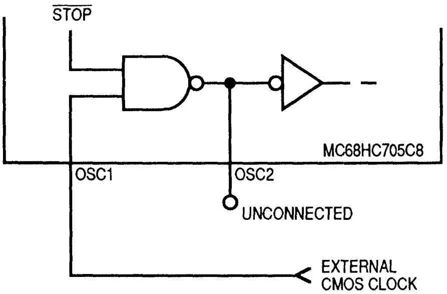
OSC1,OSC2

The MC68HC705C8 can accept either a crystal, ceramic resonator, or external input to control the internal oscillator. The internal processor clock is derived by dividing the oscillator frequency (f_{OSC}) by two.

The circuit shown in Figure 3-3(a) is recommended when using a crystal. The internal oscillator is designed to interface with an AT-cut parallel resonant quartz crystal or a ceramic resonator up to 4 MHz. The crystal and components should be mounted as close as possible to the input pins to minimize output distortion and startup stabilization time.



(a) Crystal/Ceramic Resonator Oscillator Connections



(b) External Clock Source Connections

Figure 3-3. Oscillator Connections

A ceramic resonator may be used in place of the crystal in cost-sensitive applications. The circuit in Figure 3-3(a) is recommended when using a ceramic resonator or a crystal. The manufacturer of the particular ceramic resonator being considered should be consulted for specific information.

An external clock may be applied to the OSC1 input with the OSC2 pin not connected, as shown in Figure 3-3(b).

PA7–PA0

These eight I/O lines comprise port A. Each port A pin can be software programmed to act as an input or output.

PB7–PB0

These eight lines comprise port B. Each port B pin can be software programmed to act as an input or output.

PC7–PC0

These eight lines comprise port C. Each port C pin can be software programmed to act as an input or output.

PD5–PD0, PD7

These seven lines comprise port D. During power-on or reset, these seven pins are configured as inputs. When the SPI system is enabled, four of these lines, MISO/PD2, MOSI/PD3, SCK/PD4, and SS/PD5, are used by the SPI system. When the SCI receiver is enabled, the PD0/RDI pin becomes the receive data input to the SCI. When the SCI transmitter is enabled, the PD1/TDO pin becomes the transmit data output for the SCI.

3.2.2 Typical Basic Connections

There are MCU basic connections that can be used as the starting point for any application to minimize the time required to create a prototype system.

Figure 3-4 is the schematic diagram for a simple MC68HC705C8 system. This circuit can be used as the basis for any MC68HC705C8 application. In most cases, the circuitry for the power supply and oscillator can be used as shown in this diagram. All unused inputs are terminated in an appropriate manner.

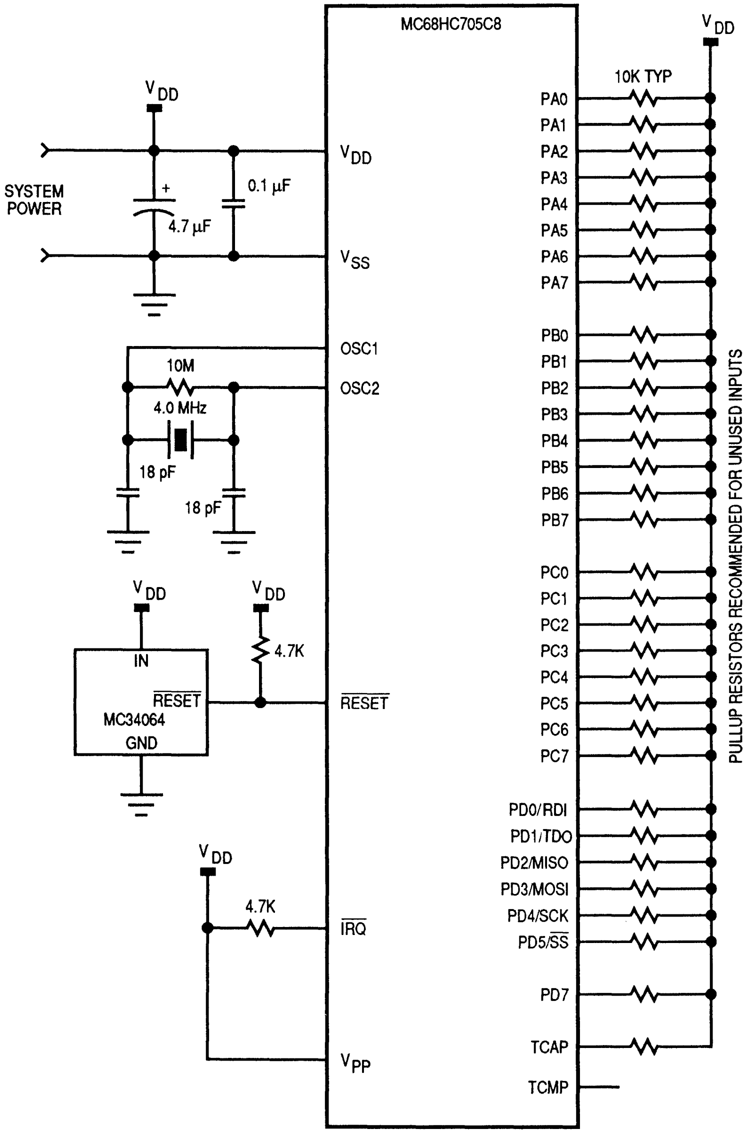


Figure 3-4. Typical Basic Connections

3.3 ON-CHIP MEMORY

The MC68HC705C8 memory includes 176 to 304 bytes of random-access memory (RAM), 240 bytes of read-only memory (ROM), and 7600 to 7744 bytes of programmable memory (EPROM or OTPROM).

3.3.1 Memory Types

RAM means that any word in the memory may be accessed without having to go through all the other words to get to it. RAM is a volatile form of memory in that all the memory content is lost when the power is removed from the chip. RAM contents may be retained by keeping at least 2 volts on V_{DD}. Power requirements in this standby mode are very small.

ROM is very similar to RAM except, unlike RAM, it is not possible to change the contents of ROM after it is manufactured. This type memory is useful only for storage of information or programs.

The special bootstrap mode allows programs to be downloaded through the on-chip serial communications interface (SCI) into internal RAM to be executed. The bootloaded program is used for a variety of tasks such as loading calibration values into internal EPROM or performing diagnostics on a finished module.

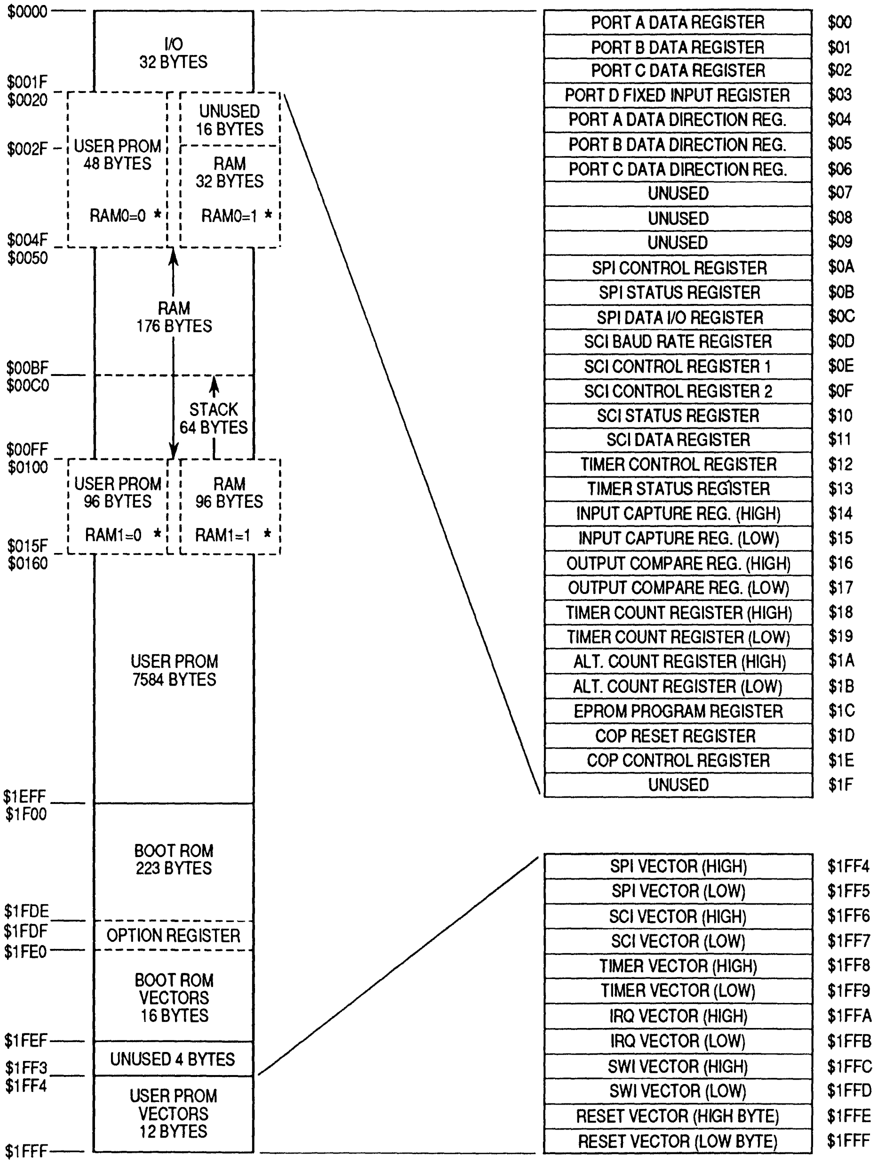
The MC68HC705C8 on-chip ROM is called the bootloader ROM. This ROM controls the loading process of the special bootstrap mode.

Erasable programmable ROM (EPROM) is nonvolatile memory that can be programmed in the field by the user. Nonvolatile memories retain their contents even when no power is applied. Once it has been programmed, the EPROM cannot be written into, but it can be read from as many times as necessary. However, EPROM can be erased by ultraviolet light and reprogrammed.

OTPROM is the same as EPROM except it can be programmed only once and cannot be erased.

3.3.2 Memory Map

The MC68HC705C8 MCU contains four selectable memory configurations as shown in Figure 3-5. The memory configurations are accessed via the option



PORT A DATA REGISTER	$00
PORT B DATA REGISTER	$01
PORT C DATA REGISTER	$02
PORT D FIXED INPUT REGISTER	$03
PORT A DATA DIRECTION REG.	$04
PORT B DATA DIRECTION REG.	$05
PORT C DATA DIRECTION REG.	$06
UNUSED	$07
UNUSED	$08
UNUSED	$09
SPI CONTROL REGISTER	$0A
SPI STATUS REGISTER	$0B
SPI DATA I/O REGISTER	$0C
SCI BAUD RATE REGISTER	$0D
SCI CONTROL REGISTER 1	$0E
SCI CONTROL REGISTER 2	$0F
SCI STATUS REGISTER	$10
SCI DATA REGISTER	$11
TIMER CONTROL REGISTER	$12
TIMER STATUS REGISTER	$13
INPUT CAPTURE REG. (HIGH)	$14
INPUT CAPTURE REG. (LOW)	$15
OUTPUT COMPARE REG. (HIGH)	$16
OUTPUT COMPARE REG. (LOW)	$17
TIMER COUNT REGISTER (HIGH)	$18
TIMER COUNT REGISTER (LOW)	$19
ALT. COUNT REGISTER (HIGH)	$1A
ALT. COUNT REGISTER (LOW)	$1B
EPROM PROGRAM REGISTER	$1C
COP RESET REGISTER	$1D
COP CONTROL REGISTER	$1E
UNUSED	$1F

SPI VECTOR (HIGH)	$1FF4
SPI VECTOR (LOW)	$1FF5
SCI VECTOR (HIGH)	$1FF6
SCI VECTOR (LOW)	$1FF7
TIMER VECTOR (HIGH)	$1FF8
TIMER VECTOR (LOW)	$1FF9
IRQ VECTOR (HIGH)	$1FFA
IRQ VECTOR (LOW)	$1FFB
SWI VECTOR (HIGH)	$1FFC
SWI VECTOR (LOW)	$1FFD
RESET VECTOR (HIGH BYTE)	$1FFE
RESET VECTOR (LOW BYTE)	$1FFF

* Refer to **3.10.4 OPTION REGISTER** for an explanation of software-selectable memory configurations.

Figure 3-5. MC68HC705C8 Memory Map

register ($1FDF) RAM0 and RAM1 bits. During reset, the RAM0 and RAM1 control bits are forced to 0. RAM0 and RAM1 bit states determine the amount of RAM and PROM, which can be selected as follows:

RAM0	RAM1	RAM Bytes	PROM Bytes
0	0	176	7744
1	0	208	7696
0	1	272	7648
1	1	304	7600

3.4 CENTRAL PROCESSOR UNIT

The MC68HC705C8 CPU is responsible for executing all software instructions in their programmed sequence for a specific application.

The CPU block diagram is shown in Figure 3-6.

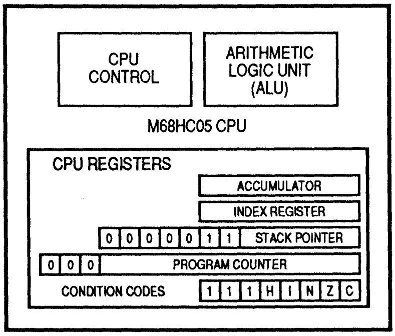

CPU CONTROL
ARITHMETIC LOGIC UNIT (ALU)
M68HC05 CPU
CPU REGISTERS
ACCUMULATOR
INDEX REGISTER
0 0 0 0 0 1 1 STACK POINTER
0 0 0 PROGRAM COUNTER
CONDITION CODES 1 1 1 H I N Z C

Figure 3-6. M68HC05 CPU Block Diagram

3.4.1 Registers

The CPU contains five registers as shown in Figure 3-7. Registers in the CPU are memories inside the microprocessor (not part of the memory map).

Accumulator (A)

The accumulator is an 8-bit general-purpose register used to hold operands, results of the arithmetic calculations, and data manipulations. It is also directly accessible to the CPU for nonarithmetic operations. The accumulator is used during the execution of a program when the contents

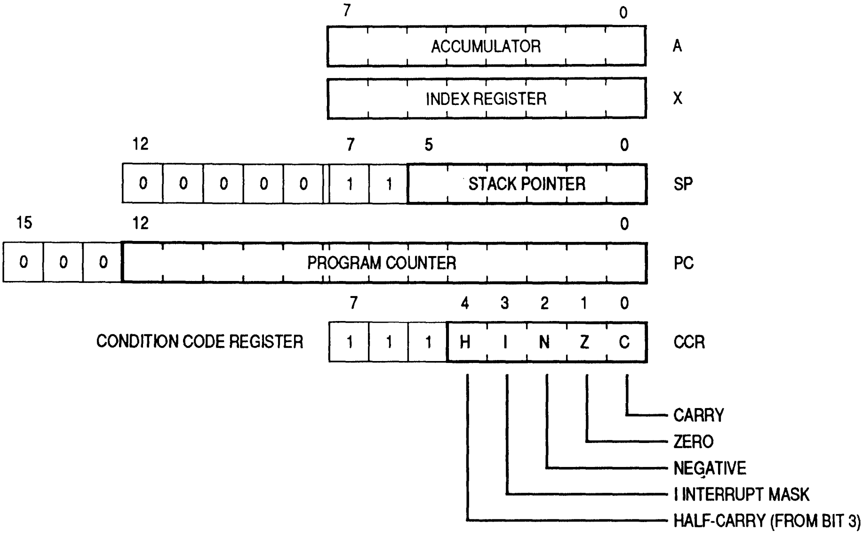


Figure 3-7. Programming Model

of some memory location are loaded into the accumulator. Also, the store instruction causes the contents of the accumulator to be stored at some prescribed memory location.

7
0
ACCUMULATOR
A

Index Register (X)

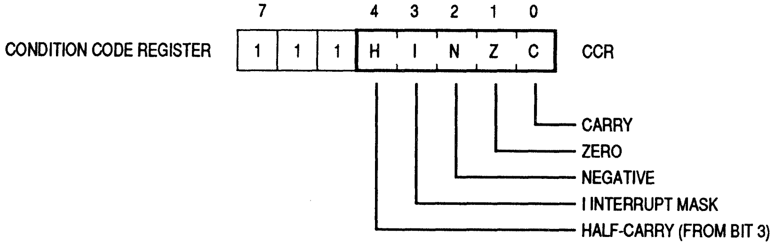
The index register is used for indexed modes of addressing or may be used as an auxiliary accumulator. This 8-bit register can be loaded either directly or from memory, have its contents stored in memory, or its contents can be compared to memory.

In indexed instructions, the X register provides an 8-bit value that is added to an instruction-provided value to create an effective address. The instruction-provided value can be 0, 1, or 2 bytes long.

7
0
INDEX REGISTER
X

Condition Code Register (CCR)

The condition code register contains five status indicators that reflect the results of arithmetic and other operations of the CPU. The five flags are half-carry (H), negative (N), zero (Z), overflow (V), and carry/borrow (C).



Half-Carry Bit (H) — The half-carry flag is used for binary-coded decimal (BCD) arithmetic operations and is affected by the ADD or ADC addition instructions. The H bit is set to a one when a carry occurs between bits 3 and 4.

Interrupt Mask Bit (I) — The interrupt mask bit disables all maskable interrupt sources when the I bit is set. Clearing this bit enables the interrupts. When any interrupt occurs, the I bit is automatically set after the registers are stacked but before the interrupt vector is fetched.

If an external interrupt occurs while the I bit is set, the interrupt is latched and processed after the I bit is cleared; therefore, no interrupts from the $\overline{\text{IRQ}}$ pin are lost because of the I bit being set.

After an interrupt has been serviced, a return from interrupt (RTI) instruction causes the registers to be restored to their previous values. Normally, the I bit would be zero after an RTI was executed. After any reset, I is set and can only be cleared by a software instruction.

Negative (N) — The N bit is set to one when the result of the last arithmetic, logical, or data manipulation is negative (bit 7 of the MSB in the result is a logic one).

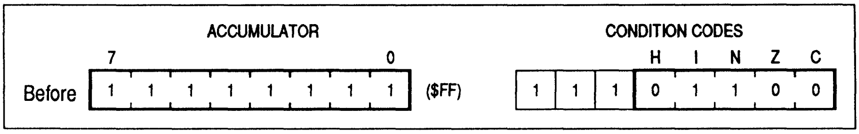
The N bit has other uses. By assigning an often-tested flag bit to the MSB of a register or memory location, you can test this bit simply by loading the accumulator with the contents of that location.

Zero (Z) — The Z bit is set to one when the result of the last arithmetic, logical, or data manipulation is zero.

Carry/Borrow (C) — The C bit is used to indicate whether or not there was a carry from an addition or a borrow as a result of a subtraction. Shift and rotate instructions operate with and through the carry bit to facilitate multiple word shift operations. This bit is also affected during bit test and branch instructions.

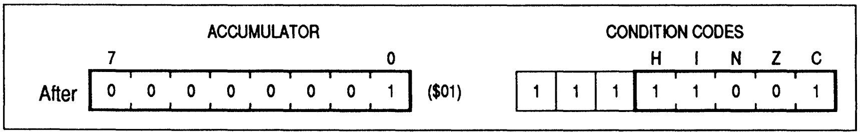
The following illustration is an example of the way condition code bits are affected by arithmetic operations.

Assume Initial Values in Accumulator and Condition Codes:



Execute the Following Instruction:

```
---- AB 02                ADD    #2     Add 2 to Accumulator
```



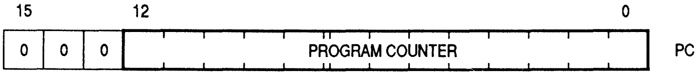
Condition Codes and Accumulator Reflect the Results of the Add Instruction:

H - Set because there was a carry from bit 3 to bit 4 of the accumulator.
I - No change.
N - Clear because result is not negative (bit 7 of accumulator is 0).
Z - Clear because result is not zero.
C - Set because there was a carry out of bit 7 of the accumulator.

The H bit is not useful after this operation because the accumulator was not a valid BCD value before the operation.

Program Counter (PC)

The program counter is a 13-bit register that contains the address of the next instruction or instruction operand to be fetched by the processor.

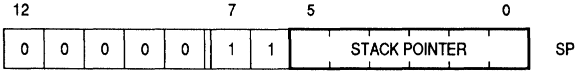


Normally, the program counter advances one memory location at a time as instructions and instruction operands are fetched.

Jump, branch, and interrupt operations cause the program counter to be loaded with a memory address other than that of the next sequential location.

Stack Pointer (SP)

The stack pointer is a 13-bit register that contains the address of the next free location on the stack. During an MCU reset or the reset stack pointer (RSP) instruction, the stack pointer is set to location $00FF. The stack pointer is then decremented as data is pushed onto the stack and incremented as data is pulled from the stack.



When accessing memory, the seven MSBs of the SP are permanently set to 0000011. These seven bits are appended to the six LSB bits to produce an address within the range of $00FF to $00CO. Subroutines and interrupts may use up to 64 (decimal) locations. If 64 locations are exceeded, the stack pointer wraps around and loses the previously stored information. A subroutine call occupies two locations on the stack; an interrupt uses five locations.

3.4.2 Arithmetic/Logic Unit (ALU)

The arithmetic/logic unit (ALU) is used to perform the arithmetic and logical operations defined by the instruction set.

The various binary arithmetic operations circuits decode the instruction in the instruction register and set up the ALU for the desired function. Most binary arithmetic is based on the addition algorithm, and subtraction is carried out as negative addition. Multiplication is not performed as a discrete instruction but as a chain of addition and shift operations within the ALU under control of CPU control logic. The multiply instruction (MUL) requires 11 internal processor cycles to complete this chain of operations.

3.4.3 CPU Control

The CPU control circuitry sequences the logic elements of the ALU to carry out the required operations.

3.4.4 Resets

Reset is used to force the MCU system to a known starting address. Peripheral systems and many control and status bits are also forced to a known state as a result of reset.

The following four conditions can cause reset in the MC68HC705C8 MCU:

1) External, active-low input signal on the $\overline{\text{RESET}}$ pin.
2) Internal power-on reset (POR) condition.
3) Internal computer operating properly (COP) watchdog system reset condition.
4) Internal clock monitor reset condition.

3.4.4.1 POWER-ON RESET. The power-on reset occurs when a positive transition is detected on V_{DD}. The power-on reset is used strictly for power turn-on conditions and should not be used to detect any drops in the power supply voltage. There is no provision for a power-down reset.

The power-on circuitry provides for a 4064 cycle delay from the time that the oscillator becomes active. If the external $\overline{\text{RESET}}$ pin is low at the end of the 4064 delay timeout, the processor remains in the reset condition until $\overline{\text{RESET}}$ goes high.

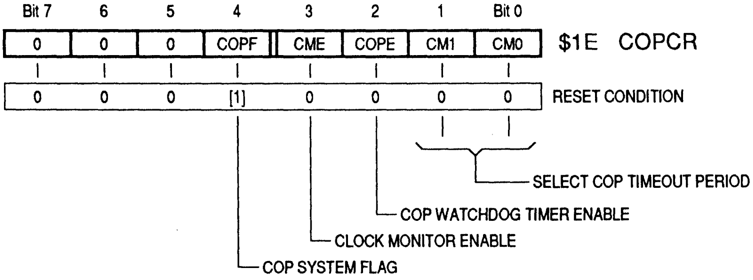
The following internal actions occur as the result of any MCU reset:

1) All data direction registers are cleared to zero (input).
2) Stack pointer configured to $00FF.
3) I bit in the condition code register to logic one.
4) External interrupt latch cleared.
5) SCI disabled (serial control bits TE=0 and RE=0). Other SCI bits cleared by reset include: TIE, TCIE, RIE, ILIE, RWU, SBK, RDRF, IDLE, OR, NF, and FE.
6) Serial status bits TDRE and TC set.
7) SCI prescaler and rate control bits SCP0, SCP1 cleared.
8) SPI disable (serial output enable control bit SPE=0). Other SPI bits cleared by reset include: SPIE, MSTR, SPIF, WCOL, and MODF.
9) All serial interrupt enable bits cleared (SPIE, TIE, and TCIE).
10) SPI system configured as slave (MSTR=0).
11) Timer prescaler reset to zero state.
    Timer counter configured to $FFFC.
    Timer output compare (TCMP) bit reset to zero.
    All timer interrupt enable bits cleared (ICIE, OCIE, and TOIE) to disable timer interrupts.
    The OLVL timer bit is also cleared by reset.
12) STOP latch cleared.
13) WAIT latch cleared.
14) Internal address bus forced to restart vector (on exit from reset, upper byte of program counter is loaded from $1FFE, and lower byte of program counter is loaded from $1FFF).

3.4.4.2 COMPUTER OPERATING PROPERLY (COP) WATCHDOG TIMER RESET. The COP watchdog timer system is intended to detect software errors. When the COP is being used, software is responsible for keeping a free-running watchdog timer from timing out. If the watchdog timer times out, it is an indication that software is no longer being executed in the intended sequence; thus, a system reset is initiated.

Since the COP timer relies on the internal bus clock in order to detect a software failure, a clock monitor is also included to guard against a failure of the clock. When the COP timer is enabled, the clock monitor should also be enabled since the COP timer cannot detect failures of the internal bus clock.

The COP control register ($1E), as shown below, is used to control the COP watchdog timer and clock monitor functions.



Bit 7	6	5	4	3	2	1	Bit 0	
0	0	0	COPF	CME	COPE	CM1	CM0	$1E COPCR
0	0	0	[1]	0	0	0	0	RESET CONDITION

[1] - Cleared on external or POR reset, set on COP or clock monitor fail resets.

COPF — Computer Operating Properly Flag
1 = COP or clock monitor reset has occurred
0 = No COP or clock monitor reset has occurred
Reading the COP control register clears COPF.

CME — Clock Monitor Enable
1 = Clock monitor enabled
0 = Clock monitor disabled
CME is readable and writable at any time.

COPE — Computer Operating Properly Enable
1 = COP timeout enabled
0 = COP timeout disabled

CM1, CM0 — Computer Operating Properly Mode
These two bits are used to select the COP watchdog timeout period (see Table 3-1).

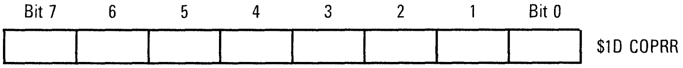
The actual timeout period is dependent on the system bus clock frequency, but, for reference purposes, Table 3-1 shows the relationship between the CM1 and CM0 select bits and the COP timeout period for various system

Table 3-1. COP Timeout Period versus CM1 and CM0

CM1	CM0	$E/2^{15}$ Div. By	XTAL=4.0 MHz E=2.0 MHz Timeout	XTAL=3.5796 E=1.7897 MHZ Timeout	XTAL=2.0 MHz E=1.0 MHz Timeout	XTAL=1.0 MHz E=0.5 MHz Timeout
0	0	1	16.38 ms	18.31 ms	32.77 ms	65.54 ms
0	1	4	65.54 ms	73.24 ms	131.07 ms	262.14 ms
1	0	16	262.14 ms	292.95 ms	524.29 ms	1.048 s
1	1	64	1.048 s	1.172 s	2.097 s	4.194 s

clock frequencies ("E" stands for the system bus clock). The default reset condition of the COP mode bits (CM1 and CM0) is cleared, which corresponds to the shortest timeout period.

The COP reset register ($1D) is used to keep the COP watchdog timer from timing out.



The sequence required to reset the COP watchdog timer is:
1) Write $55 to the COP reset register at location $1D.
2) Write $AA to the same address location.

Both write operations must occur in the correct order prior to timeout, but any number of instructions may be executed between the two write operations. The elapsed time between adjacent software reset sequences must never be greater than the COP timeout period.

Upon detection of a timeout condition, the COP watchdog timer (if enabled by COPE=1) will cause a system reset to be generated. This reset is issued to the external system via the bidirectional $\overline{\text{RESET}}$ pin for four bus cycles.

3.4.4.3 CLOCK MONITOR RESET. When a clock failure is detected by the clock monitor (and CME=1), a system reset will be generated.

When CME is set, the clock monitor detects the absence of the internal bus clock for more than a certain period of time. When CME is cleared, the clock monitor is disabled. The timeout period is dependent on processing parameters and will be between 5 and 100 μs. Thus, a bus clock rate of 200 kHz or more will never cause a clock monitor failure, and a bus clock rate of 10 kHz or less will definitely cause a clock monitor reset.

A clock monitor reset is issued to the external system via the bidirectional $\overline{\text{RESET}}$ pin for four bus cycles. The clock monitor does not have a separate reset vector.

Special considerations are needed when using the STOP instruction with the clock monitor. Since the STOP instruction causes the clocks to be halted, the clock monitor will generate a reset sequence (if enabled by CME = 1) at the time the STOP instruction is entered.

3.4.5 Addressing Modes

The power of any computer lies in its ability to access memory. The addressing modes of the CPU provide that capability. The addressing modes define the manner in which an instruction is to obtain the data required for its execution. Because of different addressing modes, an instruction may access the operand in one of up to six different ways. In this manner, the addressing modes expand the basic 62 M68HC05 Family instructions into 210 distinct opcodes.

The M68HC05 addressing modes that are used to reference memory are inherent, immediate, extended, direct, indexed (no offset, 8-bit offset, and 16-bit offset), and relative. One- and two-byte direct addressing instructions access all data bytes in most applications. Extended addressing uses three-byte instructions to reach data anywhere in memory space. The various addressing modes make it possible to locate data tables, code conversion tables, and scaling tables anywhere in the memory space. Short indexed accesses are single-byte instructions; whereas, the longest instructions (three bytes) permit accessing tables anywhere in memory.

A general description and examples of the various modes of addressing are provided in the following paragraphs. The term effective address (EA) is used to indicate the memory address where the argument for an instruction is fetched or stored. More details on addressing modes and a description of each instruction is available in Appendix A.

The information provided in the program assembly examples uses several symbols to identify the various types of numbers that occur in a program. These symbols include:

1. A blank or no symbol indicates a decimal number.
2. A $ immediately preceding a number indicates it is a hexadecimal number; e.g., $24 is 24 in hexadecimal or the equivalent of 36 in decimal.

3. A # indicates immediate operand and the number is found in the location following the opcode. A variety of symbols and expressions can be used following the character # sign. Since not all assemblers use the same syntax rules and special characters, refer to the documentation for the particular assembler that will be used.

Prefix	**Definition**
None	Decimal
$	Hexadecimal
@	Octal
%	Binary
'	Single ASCII Character

For each addressing mode, an example instruction is explained in detail. These explanations describe what happens in the CPU during each processor clock cycle of the instruction. Numbers in square brackets [ ] refer to a specific CPU clock cycle.

3.4.5.1 INHERENT ADDRESSING MODE. In inherent addressing mode, all information required for the operation is already inherently known to the CPU, and no external operand from memory or from the program is needed. The operands (if any) are only the index register and accumulator. These are always one byte instructions.

Example Program Listing:

```
0200 4c                INCA            Increment accumulator
```

Execution Sequence:
$0200 $4C [1], [2], [3]

Explanation:
[1] CPU reads opcode $4C — increment accumulator
[2], [3] CPU reads accumulator value, adds one to it, stores the new value in the accumulator, and adjusts condition code flag bits as necessary.

The following is a list of all M68HC05 instructions that can use the inherent addressing mode.

Instruction	Mnemonic
Arithmetic Shift Left	ASLA, ASLX
Arithmetic Shift Right	ASRA, ASRX
Clear Carry Bit	CLC
Clear Interrupt Mask Bit	CLI
Clear	CLRA, CLRX
Complement	COMA, COMX
Decrement	DECA, DECX
Increment	INCA, INCX
Logical Shift Left	LSLA, LSLX
Logical Shift Right	LSRA, LSRX
Multiply	MUL
Negate	NEGA, NEGX
No Operation	NOP
Rotate Left thru Carry	ROLA, ROLX
Rotate Right thru Carry	RORA, RORX
Reset Stack Pointer	RSP
Return from Interrupt	RTI
Return from Subroutine	RTS
Set Carry Bit	SEC
Set Interrupt Mask Bit	SEI
Enable IRQ, Stop Oscillator	STOP
Software Interrupt	SWI
Transfer Accumulator to Index Register	TAX
Test for Negative or Zero	TSTA, TSTX
Transfer Index Register to Accumulator	TXA
Enable Interrupt, Halt Processor	WAIT

3.4.5.2 IMMEDIATE ADDRESSING MODE. In the immediate addressing mode, the operand is contained in the byte immediately following the opcode. This mode is used to hold a value or constant which is known at the time the program is written and which is not changed during program execution. These are two-byte instructions, one for the opcode and one for the immediate data byte.

Example Program Listing:

```
0200 a6 02   LDA  #$02  Load accumulator w/ immediate value
```

Execution Sequence:

```
$0200     $A6     [1]
$0201     $02     [2]
```

Explanation:

[1] CPU reads opcode $A6 — load accumulator with the value immediately following the opcode.

[2] CPU then reads the immediate data $02 from location $0201 and loads $02 into the accumulator.

The following is a list of all M68HC05 instructions that can use the immediate addressing mode.

Instruction	Mnemonic
Add with Carry	ADC
Add	ADD
Logical AND	AND
Bit Test Memory with Accumulator	BIT
Compare Accumulator with Memory	CMP
Compare Index Register with Memory	CPX
Exclusive OR Memory with Accumulator	EOR
Load Accumulator from Memory	LDA
Load Index Register from Memory	LDX
Inclusive OR	ORA
Subtract with Carry	SBC
Subtract	SUB

3.4.5.3 EXTENDED ADDRESSING MODE. In the extended addressing mode, the address of the operand is contained in the two bytes following the opcode. Extended addressing references any location in the MCU memory space including I/O, RAM, ROM, and EPROM. Extended addressing mode instructions are three bytes, one for the opcode and two for the address of the operand.

Example Program Listing:

```
0200 c6 06 e5 LDA $06E5 Load accumulator from extended addr
```

Execution Sequence:

```
$0200  $C6  [1]
$0201  $06  [2]
$0202  $E5  [3] and [4]
```

Explanation:

[1] CPU reads opcode $C6 — load accumulator using extended addressing mode.

[2] CPU then reads $06 from location $0201. This $06 is interpreted as the high-order half of an address.

[3] CPU then reads $E5 from location $0202. This $E5 is interpreted as the low-order half of an address.

[4] CPU internally appends $06 to the $E5 read to form the complete address ($06E5). The CPU then reads whatever value is contained in the location $06E5 into the accumulator.

The following is a list of all M68HC05 instructions that can use the extended addressing mode.

Instruction	**Mnemonic**
Add with Carry	ADC
Add	ADD
Logical AND	AND
Bit Test Memory with Accumulator	BIT
Compare Acumulator with Memory	CMP
Compare Index Register with Memory	CPX
Exclusive OR Memory with Accumulator	EOR
Jump	JMP
Jump to Subroutine	JSR
Load Accumulator from Memory	LDA
Load Index Register from Memory	LDX
Inclusive OR	ORA
Subtract with Carry	SBC
Store Accumulator in Memory	STA
Store Index Register in Memory	STX
Subtract	SUB

3.4.5.4 DIRECT ADDRESSING MODE. The direct addressing mode is similar to the extended addressing mode except the upper byte of the operand address is assumed to be $00. Thus, only the lower byte of the operand address needs to be included in the instruction. Direct addressing allows you to efficiently address the lowest 256 bytes in memory. This area of memory is called the direct page and includes on-chip RAM and I/O registers. Direct addressing is efficient in both memory and time. Direct addressing mode instructions are usually two bytes, one for the opcode and one for the low-order byte of the operand address.

Example Program Listing:

```
0200 b6 50  LDA   $50   Load accumulator from direct address
```

Execution Sequence:
$0200 $B6 [1]
$0201 $50 [2] and [3]

Explanation:

[1] CPU reads opcode $B6 — load accumulator using direct addressing mode.

[2] CPU then reads $50 from location $0201. This $50 is interpreted as the low-order half of an address. In direct addressing mode, the high-order half of the address is assumed to be $00.

[3] CPU internally appends $00 to the $50 read in the second cycle to form the complete address ($0050). The CPU then reads whatever value is contained in the location $0050 into the accumulator.

The following is a list of all M68HC05 instructions that can use the direct addressing mode.

Instruction	Mnemonic
Add with Carry	ADC
Add	ADD
Logical AND	AND
Arithmetic Shift Left	ASL
Arithmetic Shift Right	ASR
Clear Bit in Memory	BCLR
Bit Test Memory with Accumulator	BIT
Branch if Bit n is Clear	BRCLR
Branch if Bit n is Set	BRSET
Set Bit in Memory	BSET
Clear	CLR
Compare Accumulator with Memory	CMP
Complement	COM
Compare Index Register with Memory	CPX
Decrement	DEC
Exclusive OR Memory with Accumulator	EOR
Increment	INC
Jump	JMP
Jump to Subroutine	JSR
Load Accumulator from Memory	LDA
Load Index Register from Memory	LDX
Logical Shift Left	LSL
Logical Shift Right	LSR
Negate	NEG
Inclusive OR	ORA
Rotate Left thru Carry	ROL
Rotate Right thru Carry	ROR
Subtract with Carry	SBC
Store Accumulator in Memory	STA
Store Index Register in Memory	STX
Subtract	SUB
Test for Negative or Zero	TST

3.4.5.5 INDEXED ADDRESSING MODE. In the indexed addressing mode, the effective address is variable and depends upon two factors: 1) the current contents of the index (X) register and 2) the offset contained in the byte(s) following the opcode. Three types of indexed addressing exist in the MCU: no offset, 8-bit offset, and 16-bit offset. A good assembler should use the indexed addressing mode that requires the least number of bytes to express the offset.

3.4.5.5.1 Indexed, No Offset. In the indexed, no-offset addressing mode, the effective address of the instruction is contained in the 8-bit index register. Thus, this addressing mode can access the first 256 memory locations. These instructions are only one byte.

Example Program Listing:

```
0200 f6    LDA   ,X       Load accumulator from location
                          pointed to by index reg (no offset)
```

Execution Sequence:

```
$0200  $F6    [1],  [2],  [3]
```

Explanation:

[1] CPU reads opcode $F6 — load accumulator using indexed, no offset, addressing mode.

[2] CPU forms a complete address by adding $0000 to the contents of the index register.

[3] CPU then reads the contents of the addressed location into the accumulator.

The following is a list of all M68HC05 instructions that can use the indexed, no-offset addressing mode.

Instruction	Mnemonic
Add with Carry	ADC
Add	ADD
Logical AND	AND
Arithmetic Shift Left	ASL
Arithmetic Shift Right	ASR
Bit Test Memory with Accumulator	BIT
Clear	CLR
Compare Accumulator with Memory	CMP
Complement	COM
Compare Index Register with Memory	CPX
Decrement	DEC
Exclusive OR Memory with Accumulator	EOR
Increment	INC
Jump	JMP
Jump to Subroutine	JSR
Load Accumulator from Memory	LDA
Load Index Register from Memory	LDX
Logical Shift Left	LSL
Logical Shift Right	LSR
Negate	NEG
Inclusive OR	ORA
Rotate Left thru Carry	ROL
Rotate Right thru Carry	ROR
Subtract with Carry	SBC
Store Accumulator in Memory	STA
Store Index Register in Memory	STX
Subtract	SUB
Test for Negative or Zero	TST

3.4.5.5.2 Indexed, 8-Bit Offset. In the indexed, 8-bit offset addressing mode, the effective address is obtained by adding the contents of the byte following the opcode to the contents of the index register. This mode of addressing is useful for selecting the kth element in a 'n' element table. To use this mode, the table must begin in the lowest 256 memory locations, and may extend through the first 511 memory locations (1FE is the last location which the instruction may access). Indexed 8-bit offset addressing can be used for ROM, RAM, or I/O. This is a two-byte instruction with the offset contained in the byte following the opcode. The content of the index register (X) is not changed. The offset byte supplied in the instruction is an unsigned 8-bit integer.

Example Program Listing:

```
0200 e6 05  LDA   $5,X     Load accumulator from location
                            pointed to by index reg (X) + $05
```

Execution Sequence:

```
$0200  $E6  [1]
$0201  $05  [2],  [3],  [4]
```

Explanation:

[1] CPU reads opcode $E6 — load accumulator using indexed, 8-bit offset addressing mode.

[2] CPU then reads $05 from location $0201. This $05 is interpreted as the low-order half of a base address. The high-order half of the base address is assumed to be $00.

[3] CPU will add the value in the index register to the base address $0005. The results of this addition is the address that the CPU will use in the load accumulator operation.

[4] The CPU will then read the value from this address and load this value into the accumulator.

The following is a list of all M68HC05 instructions that can use the indexed, 8-bit offset addressing mode.

Instruction	Mnemonic
Add with Carry	ADC
Add	ADD
Logical AND	AND
Arithmetic Shift Left	ASL
Arithmetic Shift Right	ASR
Bit Test Memory with Accumulator	BIT
Clear	CLR
Compare Accumulator with Memory	CMP
Complement	COM
Compare Index Register with Memory	CPX
Decrement	DEC
Exclusive OR Memory with Accumulator	EOR
Increment	INC
Jump	JMP
Jump to Subroutine	JSR
Load Accumulator from Memory	LDA
Load Index Register from Memory	LDX
Logical Shift Left	LSL
Logical Shift Right	LSR
Negate	NEG
Inclusive OR	ORA
Rotate Left thru Carry	ROL
Rotate Right thru Carry	ROR
Subtract with Carry	SBC
Store Accumulator in Memory	STA
Store Index Register in Memory	STX
Subtract	SUB
Test for Negative or Zero	TST

3.4.5.5.3 Indexed, 16-Bit Offset. In the indexed, 16-bit offset addressing mode, the effective address is the sum of the contents of the 8-bit index register and the two bytes following the opcode. The content of the index register is not changed. These instructions are three bytes, one for the opcode and two for a 16-bit offset.

Example Program Listing:

```
0200 d6 07 00 LDA $0700,X Load accumulator from location
                          pointed to by index reg (X) + $0700
```

Execution Sequence:

```
$0200  $D6  [1]
$0201  $07  [2]
$0202  $00  [3],  [4],  [5]
```

Explanation:

[1] CPU reads opcode $D6 — load accumulator using indexed, 16-bit offset addressing mode.

[2] CPU then reads $07 from location $0201. This $07 is interpreted as the high-order half of a base address.

[3] CPU then reads $00 from location $0202. This $00 is interpreted as the low-order half of a base address.

[4] CPU will add the value in the index register to the base address $0700. The results of this addition is the address that the CPU will use in the load accumulator operation.

[5] The CPU will then read the value from this address and load this value into the accumulator.

The following is a list of all M68HC05 instructions that can use the indexed, 16-bit offset addressing mode.

Instruction	Mnemonic
Add with Carry	ADC
Add	ADD
Logical AND	AND
Bit Test Memory with Accumulator	BIT
Compare Accumulator with Memory	CMP
Compare Index Register with Memory	CPX
Exclusive OR Memory with Accumulator	EOR
Jump	JMP
Jump to Subroutine	JSR
Load Accumulator from Memory	LDA
Load Index Register from Memory	LDX
Inclusive OR	ORA
Subtract with Carry	SBC
Store Accumulator in Memory	STA
Store Index Register In Memory	STX
Subtract	SUB

3.4.5.6 RELATIVE ADDRESSING MODE. The relative addressing mode is used only for branch instructions. Branch instructions, other than the branching versions of bit-manipulation instructions, generate two machine-code bytes: one for the opcode and one for the relative offset. Because it is desirable to branch in either direction, the offset byte is a signed twos-complement offset with a range of −127 to +128 bytes (with respect to the address of the instruction immediately following the branch instruction). If the branch condition is true, the contents of the 8-bit signed byte following the opcode (offset) are added to the contents of the program counter to form the effective branch address; otherwise, control proceeds to the instruction immediately following the branch instruction.

A programmer specifies the destination of a branch as an absolute address (or label which refers to an absolute address). The Motorola assembler calculates the 8-bit signed relative offset, which is placed after the branch opcode in memory.

Example Program Listing:

```
0200 27 rr           BEQ   DEST     Branch to DEST if Z=1
                                    (branch if equal or zero)
```

Execution Sequence:

$0200 $27 [1]
$0201 $rr [2], [3]

Explanation:

[1] CPU reads opcode $27 — branch if Z = 1, (relative addressing mode).
[2] CPU reads the offset, $rr.
[3] CPU internally tests the state of the Z bit and causes a branch if Z is set.

The following is a list of all M68HC05 instructions that can use the relative addressing mode.

Instruction	Mnemonic
Branch if Carry Clear	BCC
Branch is Carry Set	BCS
Branch if Equal	BEQ
Branch if Half-Carry Clear	BHCC
Branch if Half-Carry Set	BHCS
Branch if Higher	BHI
Branch if Higher or Same	BHS
Branch if Interrupt Line is High	BIH
Branch if Interrupt Line is Low	BIL
Branch if Lower	BLO
Branch if Lower or Same	BLS
Branch if Interrupt Mask is Clear	BMC
Branch if Minus	BMI
Branch if Interrupt Mask Bit is Set	BMS
Branch if Not Equal	BNE
Branch if Plus	BPL
Branch Always	BRA
Branch if Bit n is Clear	BRCLR
Branch if Bit n is Set	BRSET
Branch Never	BRN
Branch to Subroutine	BSR

3.4.5.7 BIT TEST AND BRANCH INSTRUCTIONS. These instructions use direct addressing mode to specify the location being tested and relative addressing to specify the branch destination. This applications guide treats these instructions as direct addressing mode instructions. Some older Motorola documents call the addressing mode of these instructions BTB for bit test and branch.

3.4.5.8 INSTRUCTIONS ORGANIZED BY TYPE. Tables 3-2 through 3-5 show the MC68HC05 instruction set displayed by instruction type.

Table 3-2. Register/Memory Instructions

		Addressing Modes																	
		Immediate			Direct			Extended			Indexed (No Offset)			Indexed (8-Bit Offset)			Indexed (16-Bit Offset)		
Function	Mnem.	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles
Load A from Memory	LDA	A6	2	2	B6	2	3	C6	3	4	F6	1	3	E6	2	4	D6	3	5
Load X from Memory	LDX	AE	2	2	BE	2	3	CE	3	4	FE	1	3	EE	2	4	DE	3	5
Store A in Memory	STA	—	—	—	B7	2	4	C7	3	5	F7	1	4	E7	2	5	D7	3	6
Store X in Memory	STX	—	—	—	BF	2	4	CF	3	5	FF	1	4	EF	2	5	DF	3	6
Add Memory to A	ADD	AB	2	2	BB	2	3	CB	3	4	FB	1	3	EB	2	4	DB	3	5
Add Memory and Carry to A	ADC	A9	2	2	B9	2	3	C9	3	4	F9	1	3	E9	2	4	D9	3	5
Subtract Memory	SUB	A0	2	2	B0	2	3	C0	3	4	F0	1	3	E0	2	4	D0	3	5
Subtract Memory from A with Borrow	SBC	A2	2	2	B2	2	3	C2	3	4	F2	1	3	E2	2	4	D2	3	5
AND Memory to A	AND	A4	2	2	B4	2	3	C4	3	4	F4	1	3	E4	2	4	D4	3	5
OR Memory with A	ORA	AA	2	2	BA	2	3	CA	3	4	FA	1	3	EA	2	4	DA	3	5
Exclusive OR Memory with A	EOR	A8	2	2	B8	2	3	C8	3	4	F8	1	3	E8	2	4	D8	3	5
Arithmetic Compare A with Memory	CMP	A1	2	2	B1	2	3	C1	3	4	F1	1	3	E1	2	4	D1	3	5
Arithmetic Compare X with Memory	CPX	A3	2	2	B3	2	3	C3	3	4	F3	1	3	E3	2	4	D3	3	5
Bit Test Memory with A (Logical Compare)	BIT	A5	2	2	B5	2	3	C5	3	4	F5	1	3	E5	2	4	D5	3	5
Jump Unconditional	JMP	—	—	—	BC	2	2	CC	3	3	FC	1	2	EC	2	3	DC	3	4
Jump to Subroutine	JSR	—	—	—	BD	2	5	CD	3	6	FD	1	5	ED	2	6	DD	3	7

Table 3-3. Read/Modify-Write Instructions

		Addressing Modes														
		Inherent (A)			Inherent (X)			Direct			Indexed (No Offset)			Indexed (8-Bit Offset)		
Function	Mnem.	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles	Op-code	# Bytes	# Cycles
Increment	INC	4C	1	3	5C	1	3	3C	2	5	7C	1	5	6C	2	6
Decrement	DEC	4A	1	3	5A	1	3	3A	2	5	7A	1	5	6A	2	6
Clear	CLR	4F	1	3	5F	1	3	3F	2	5	7F	1	5	6F	2	6
Complement	COM	43	1	3	53	1	3	33	2	5	73	1	5	63	2	6
Negate (2's Complement)	NEG	40	1	3	50	1	3	30	2	5	70	1	5	60	2	6
Rotate Left Thru Carry	ROL	49	1	3	59	1	3	39	2	5	79	1	5	69	2	6
Rotate Right Thru Carry	ROR	46	1	3	56	1	3	36	2	5	76	1	5	66	2	6
Logical Shift Left	LSL	48	1	3	58	1	3	38	2	5	78	1	5	68	2	6
Logical Shift Right	LSR	44	1	3	54	1	3	34	2	5	74	1	5	64	2	6
Arithmetic Shift Right	ASR	47	1	3	57	1	3	37	2	5	77	1	5	67	2	6
Test for Negative or Zero	TST	4D	1	3	5D	1	3	3D	2	4	7D	1	4	6D	2	5
Multiply	MUL	42	1	11	—	—	—	—	—	—	—	—	—	—	—	—

Table 3-4. Branch Instructions

Function	Mnemonic	Relative Addressing Mode		
		Opcode	# Bytes	# Cycles
Branch Always	BRA	20	2	3
Branch Never	BRN	21	2	3
Branch IFF Higher	BHI	22	2	3
Branch IFF Lower or Same	BLS	23	2	3
Branch IFF Carry Clear	BCC	24	2	3
Branch IFF Higher or Same (Same as BCC)	BHS	24	2	3
Branch IFF Carry Set	BCS	25	2	3
Branch IFF Lower (Same as BCS)	BLO	25	2	3
Branch IFF Not Equal	BNE	26	2	3
Branch IFF Equal	BEQ	27	2	3
Branch IFF Half-Carry Clear	BHCC	28	2	3
Branch IFF Half-Carry Set	BHCS	29	2	3
Branch IFF Plus	BPL	2A	2	3
Branch IFF Minus	BMI	2B	2	3
Branch IFF Interrupt Mask Bit is Clear	BMC	2C	2	3
Branch IFF Interrupt Mask Bit is Set	BMS	2D	2	3
Branch IFF Interrupt Line is Low	BIL	2E	2	3
Branch IFF Interrupt Line is High	BIH	2F	2	3
Branch to Subroutine	BSR	AD	2	6

Table 3-5. Control Instructions

Function	Mnemonic	Inherent		
		Opcode	# Bytes	# Cycles
Transfer A to X	TAX	97	1	2
Transfer X to A	TXA	9F	1	2
Set Carry Bit	SEC	99	1	2
Clear Carry Bit	CLC	98	1	2
Set Interrupt Mask Bit	SEI	9B	1	2
Clear Interrupt Mask Bit	CLI	9A	1	2
Software Interrupt	SWI	83	1	10
Return from Subroutine	RTS	81	1	6
Return from Interrupt	RTI	80	1	9
Reset Stack Pointer	RSP	9C	1	2
No-Operation	NOP	9D	1	2
Stop	STOP	8E	1	2
Wait	WAIT	8F	1	2

3.4.6 Instruction Set Summary

Computers use an operation code or opcode to give instructions to the CPU. The instruction set for a specific CPU is the set of all opcodes that the CPU knows how to execute. The CPU in the MC68HC705C8 MCU can understand 62 basic instructions, some of which have several variations that require separate opcodes. The M68HC05 instruction set includes 210 unique instruction opcodes.

The following table is an alphabetical listing of the M68HC05 instructions available to the user. In listing all the factors necessary to program, the table uses the following symbols:

Condition Code Symblols

H	— Half Carry (Bit 4)	↕	— Test and Set if True, (cleared otherwise)
I	— Interrupt Mask (Bit 3)		
N	— Negate (Sign Bit 2)	—	— Not Affected
Z	— Zero (Bit 1)	?	— Load CC from Stack
C	— Carry/Borrow (Bit 0)	0	— Cleared
		1	— Set

Boolean Operators

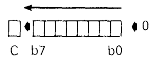
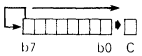
( )	— Contents of (i.e., (M) means the contents of memory location M)	+	— (inclusive) OR
		⊕	— Exclusive OR
		—	— NOT
		-	— Negation (twos complement)
←	— is loaded with, 'gets'		
●	— AND	×	— Multiplication

MPU Registers

A	— Accumulator	PC	— Program Counter
ACCA	— Accumulator	PCH	— PC High Byte
CC	— Condition Code Reg.	PCL	— PC Low Byte
X	— Index Register	SP	— Stack Pointer
M	— Any memory location (one byte)	REL	— Relative Address

Addressing Modes	(Abbreviation)	Operands	
Inherent	INH	none	
Immediate	IMM	ii	
Direct (for bit	DIR	dd	
test instructions)		dd	rr
Extended	EXT	hh	ll
Indexed 0 Offset	IX	none	
Indexed 1-Byte	IX1	ff	
Indexed 2-Byte	IX2	ee	ff
Relative	REL	rr	

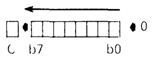
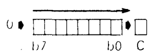
INSTRUCTION ADDRESSING MODES, AND EXECUTION TIMES (Sheet 1 of 4)

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (hexadecimal)		Bytes	Cycles	Condition Code				
				Opcode	Operand			H	I	N	Z	C
ADC (opr)	Add with Carry	ACCA ← ACCA + M + C	IMM DIR EXT IX2 IX1 IX	A9 B9 C9 D9 E9 F9	ii dd hh ll ee ff ff	2 2 3 3 2 1	2 3 4 5 4 3	↕	—	↕	↕	↕
ADD (opr)	Add	ACCA ← ACCA + M	IMM DIR EXT IX2 IX1 IX	AB BB CB DB EB FB	ii dd hh ll ee ff ff	2 2 3 3 2 1	2 3 4 5 4 3	↕	—	↕	↕	↕
AND (opr)	Logical AND	ACCA ← ACCA ● M	IMM DIR EXT IX2 IX1 IX	A4 B4 C4 D4 E4 F4	ii dd hh ll ee ff ff	2 2 3 3 2 1	2 3 4 5 4 3	—	—	↕	↕	—
ASL (opr) ASLA ASLX ASL (opr) ASL (opr)	Arithmetic Shift Left		DIR INH(A) INH(X) IX1 IX	38 48 58 68 78	dd ff	2 1 1 2 1	5 3 3 6 5	—	—	↕	↕	↕
ASR (opr) ASRA ASRX ASR (opr) ASR (opr)	Arithmetic Shift Right		DIR INH(A) INH(X) IX1 IX	37 47 57 67 77	dd ff	2 1 1 2 1	5 3 3 6 5	—	—	↕	↕	↕
BCC (rel)	Branch if Carry Clear	? C = 0	REL	24	rr	2	3	—	—	—	—	—
BCLR n, (opr)	Clear Bit n in Memory	Mn ← 0	DIR(b0) DIR(b1) DIR(b2) DIR(b3) DIR(b4) DIR(b5) DIR(b6) DIR(b7)	11 13 15 17 19 1B 1D 1F	dd dd dd dd dd dd dd dd	2 2 2 2 2 2 2 2	5 5 5 5 5 5 5 5	—	—	—	—	—
BCS (rel)	Branch if Carry Set	? C = 1	REL	25	rr	2	3	—	—	—	—	—
BEQ (rel)	Branch if Equal	? Z = 1	REL	27	rr	2	3	—	—	—	—	—
BHCC (rel)	Branch if Half Carry Clear	? H = 0	REL	28	rr	2	3	—	—	—	—	—
BHCS (rel)	Branch if Half Carry Set	? H = 1	REL	29	rr	2	3	—	—	—	—	—
BHI (rel)	Branch if Higher	? (C + Z) = 0	REL	22	rr	2	3	—	—	—	—	—
BHS (rel)	Branch if Higher or Same	? C = 0	REL	24	rr	2	3	—	—	—	—	—
BIH (rel)	Branch if $\overline{IRQ}$ Pin is High	? $\overline{IRQ}$ Pin = 1	REL	2F	rr	2	3	—	—	—	—	—
BIL (rel)	Branch if $\overline{IRQ}$ Pin is Low	? $\overline{IRQ}$ Pin = 0	REL	2E	rr	2	3	—	—	—	—	—
BIT (rel)	Bit Test Memory with A	ACCA ● M	IMM DIR EXT IX2 IX1 IX	A5 B5 C5 D5 E5 F5	ii dd hh ll ee ff ff	2 2 3 3 2 1	2 3 4 5 4 3	—	—	↕	↕	—
BLO (rel)	Branch if Lower	? C = 1	REL	25	rr	2	3	—	—	—	—	—
BLS (rel)	Branch if Lower or Same	? (C + X) = 1	REL	23	rr	2	3	—	—	—	—	—
BMC (rel)	Branch if I Bit is Clear	? I = 0	REL	2C	rr	2	3	—	—	—	—	—

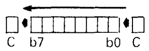
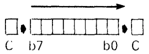
INSTRUCTION ADDRESSING MODES, AND EXECUTION TIMES (Sheet 2 of 4)

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (hexadecimal)		Bytes	Cycles	Condition Code				
				Opcode	Operand			H	I	N	Z	C
BMI (rel)	Branch if Minus	? N – 1	REL	2B	rr	2	3	—	—	—	—	—
BMS (rel)	Branch if I Bit is Set	? I – 1	REL	2D	rr	2	3	—	—	—	—	—
BNE (rel)	Branch if Not Equal	? Z – 0	REL	26	rr	2	3	—	—	—	—	—
BPL (rel)	Branch if Plus	? N – 0	REL	2A	rr	2	3	—	—	—	—	—
BRA (rel)	Branch Always	? 1 – 1	REL	20	rr	2	3	—	—	—	—	—
BRCLR n, (opr) (rel)	Branch if Bit n of M - 0	? Bit n of M – 0	DIR(b0) DIR(b1) DIR(b2) DIR(b3) DIR(b4) DIR(b5) DIR(b6) DIR(b7)	01 03 05 07 09 0B 0D 0F	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	3 3 3 3 3 3 3 3	5 5 5 5 5 5 5 5	—	—	—	—	↕
BRN (rel)	Branch Never	? 1 · 0	REL	21	rr	2	3	—	—	—	—	—
BRSET n, (opr) (rel)	Branch if Bit n of M 1	? Bit n of M · 1	DIR(b0) DIR(b1) DIR(b2) DIR(b3) DIR(b4) DIR(b5) DIR(b6) DIR(b7)	00 02 04 06 08 0A 0C 0E	dd rr dd rr dd rr dd rr dd rr dd rr dd rr dd rr	3 3 3 3 3 3 3 3	5 5 5 5 5 5 5 5	—	—	—	—	↕
BSET n, (opr)	Set Bit n in Memory	Mn ← 1	DIR(b0) DIR(b1) DIR(b2) DIR(b3) DIR(b4) DIR(b5) DIR(b6) DIR(b7)	10 12 14 16 18 1A 1C 1E	dd dd dd dd dd dd dd dd	2 2 2 2 2 2 2 2	5 5 5 5 5 5 5 5	—	—	—	—	—
BSR (rel)	Branch to Subroutine	PC ← PC + 0002 (SP) ← PCL; SP ← SP - 0001 (SP) ← PCH; SP ← SP - 0001 PC ← PC + Rel	REL	AD	rr	2	6	—	—	—	—	—
CLC	Clear C Bit	C bit ← 0	INH	98		1	2	—	—	—	—	0
CLI	Clear I Bit	I bit ← 0	INH	9A		1	2	—	0	—	—	—
CLR (opr) CLRA CLRX CLR (opr) CLR (opr)	Clear	M ← 00 A ← 00 X ← 00 M ← 00 M ← 00	DIR INH(A) INH(X) IX1 IX	3F 4F 5F 6F 7F	dd ff 	2 1 1 2 1	5 3 3 6 5	—	—	0	1	—
CMP (opr)	Compare A with Memory	ACCA - M	IMM DIR EXT IX2 IX1 IX	A1 B1 C1 D1 E1 F1	ii dd hh ll ee ff ff 	2 2 3 3 2 1	2 3 4 5 4 3	—	—	↕	↕	↕
COM (opr) COMA COMX COM (opr) COM (opr)	1's Complement	M ← $\overline{M}$ – $FF - M A ← $\overline{A}$ – $FF - A X ← $\overline{X}$ – $FF - X M ← $\overline{M}$ – $FF – M M ← $\overline{M}$ – $FF – M	DIR INH(A) INH(X) IX1 IX	33 43 53 63 73	dd ff 	2 1 1 2 1	5 3 3 6 5	—	—	↕	↕	1
CPX (opr)	Compare X with Memory	X – M	IMM DIR EXT IX2 IX1 IX	A3 B3 C3 D3 E3 F3	ii dd hh ll ee ff ff 	2 2 3 3 2 1	2 3 4 5 4 3	—	—	↕	↕	↕

INSTRUCTION ADDRESSING MODES, AND EXECUTION TIMES (Sheet 3 of 4)

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (hexadecimal)		Bytes	Cycles	Condition Code				
				Opcode	Operand			H	I	N	Z	C
DEC (opr)	Decrement	M ← M − 01	DIR	3A	dd	2	5	—	—	↕	↕	—
DECA		A ← A − 01	INH(A)	4A		1	3					
DECX	DEX (same as DECX)	X ← X − 01	INH(X)	5A		1	3					
DEC (opr)		M ← M − 01	IX1	6A	ff	2	6					
DEC (opr)		M ← M − 01	IX	7A		1	5					
EOR (opr)	Exclusive OR A with Memory	ACCA ← ACCA ⊕ M	IMM	A8	ii	2	2	—	—	↕	↕	—
			DIR	B8	dd	2	3					
			EXT	C8	hh ll	3	4					
			IX2	D8	ee ff	3	5					
			IX1	E8	ff	2	4					
			IX	F8		1	3					
INC (opr)	Increment	M ← M + 01	DIR	3C	dd	2	5	—	—	↕	↕	—
INCA		A ← A + 01	INH(A)	4C		1	3					
INCX	INX (same as INCX)	X ← X + 01	INH(X)	5C		1	3					
INC (opr)		M ← M + 01	IX1	6C	ff	2	6					
INC (opr)		M ← M + 01	IX	7C		1	5					
JMP (opr)	Jump	PC ← effective address	DIR	BC	dd	2	2	—	—	—	—	—
			EXT	CC	hh ll	3	3					
			IX2	DC	ee ff	3	4					
			IX1	EC	ff	2	3					
			IX	FC		1	2					
JSR (opr)	Jump to Subroutine	PC ← PC + n (n = 1, 2, or 3)	DIR	BD	dd	2	5	—	—	—	—	—
		(SP) ← PCL; SP ← SP − 0001	EXT	CD	hh ll	3	6					
		(SP) ← PCH; SP ← SP − 0001	IX2	DD	ee ff	3	7					
		PC ← effective address	IX1	ED	ff	2	6					
			IX	FD		1	5					
LDA (opr)	Load A from Memory	ACCA ← M	IMM	A6	ii	2	2	—	—	↕	↕	—
			DIR	B6	dd	2	3					
			EXT	C6	hh ll	3	4					
			IX2	D6	ee ff	3	5					
			IX1	E6	ff	2	4					
			IX	F6		1	3					
LDX (opr)	Load X from Memory	X ← M	IMM	AE	ii	2	2	—	—	↕	↕	—
			DIR	BE	dd	2	3					
			EXT	CE	hh ll	3	4					
			IX2	DE	ee ff	3	5					
			IX1	EE	ff	2	4					
			IX	FE		1	3					
LSL (opr)	Logical Shift Left		DIR	38	dd	2	5	—	—	↕	↕	↕
LSLA			INH(A)	48		1	3					
LSLX			INH(X)	58		1	3					
LSL (opr)			IX1	68	ff	2	6					
LSL (opr)			IX	78		1	5					
LSR (opr)	Logical Shift Right		DIR	34	dd	2	5	—	—	0	↕	↕
LSRA			INH(A)	44		1	3					
LSRX			INH(X)	54		1	3					
LSR (opr)			IX1	64	ff	2	6					
LSR (opr)			IX	74		1	5					
MUL	Unsigned Multiply	X:A ← X • A	INH	42		1	11	0	—	—	—	0
NEG (opr)	Negate (2's Complement)	M ← − M (i.e. 00 − M)	DIR	30	dd	2	5	—	—	↕	↕	↕
NEGA		A ← − A	INH(A)	40		1	3					
NEGX		X ← − X	INH(X)	50		1	3					
NEG (opr)		M ← − M	IX1	60	ff	2	6					
NEG (opr		M ← − M	IX	70		1	5					
NOP	No Operation		INH	9D		1	2	—	—	—	—	—
ORA (opr)	Inclusive OR	ACCA ← ACCA + M	IMM	AA	ii	2	2	—	—	↕	↕	—
			DIR	BA	dd	2	3					
			EXT	CA	hh ll	3	4					
			IX2	DA	ee ff	3	5					
			IX1	EA	ff	2	4					
			IX	FA		1	3					

INSTRUCTION ADDRESSING MODES, AND EXECUTION TIMES (Sheet 4 of 4)

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (hexadecimal)		Bytes	Cycles	Condition Code				
				Opcode	Operand			H	I	N	Z	C
ROL (opr)	Rotate Left through Carry		DIR	39	dd	2	5	—	—	↕	↕	↕
ROLA			INH(A)	49		1	3					
ROLX			INH(X)	59		1	3					
ROL (opr)			IX1	69	ff	2	6					
ROL (opr			IX	79		1	5					
ROR (opr)	Rotate Right through Carry		DIR	36	dd	2	5	—	—	↕	↕	↕
RORA			INH(A)	46		1	3					
RORX			INH(X)	56		1	3					
ROR (opr)			IX1	66	ff	2	6					
ROR (opr)			IX	76		1	5					
RSP	Reset Stack Pointer	SP ← $00FF	INH	9C		1	2	—	—	—	—	—
RTI	Return from Interrupt	SP ← SP + 0001; CC ← (SP)	INH	80		1	9	(From Stack)				
		SP ←SP + 0001; ACCA← (SP)						↕	↕	↕	↕	↕
		SP ← SP + 0001; X ← (SP)										
		SP ← SP + 0001; PCH ← (SP)										
		SP ← SP + 0001; PCL ← (SP)										
RTS	Return from Subroutine	SP ← SP + 0001; PCH ← (SP)	INH	81		1	6	—	—	—	—	—
		SP ← SP + 0001; PCL ← (SP)										
SBC (opr)	Subtract with Carry	ACCA ← ACCA − M − C	IMM	A2	ii	2	2	—	—	↕	↕	↕
			DIR	B2	dd	2	3					
			EXT	C2	hh ll	3	4					
			IX2	D2	ee ff	3	5					
			IX1	E2	ff	2	4					
			IX	F2		1	3					
SEC	Set C Bit	C bit ← 1	INH	99		1	2	—	—	—	—	1
SEI	Set I Bit	I bit ← 1	INH	9B		1	2	—	1	—	—	—
STA (opr)	Store A in Memory	M ← ACCA	DIR	B7	dd	2	4	—	—	↕	↕	—
			EXT	C7	hh ll	3	5					
			IX2	D7	ee ff	3	6					
			IX1	E7	ff	2	5					
			IX	F7		1	4					
STOP	Enable $\overline{IRQ}$, Stop Oscillator		INH	8E		1	2	—	0	—	—	—
STX (opr)	Store X in Memory	M ← X	DIR	BF	dd	2	4	—	—	↕	↕	0
			EXT	CF	hh ll	3	5					
			IX2	DF	ee ff	3	6					
			IX1	EF	ff	2	5					
			IX	FF		1	4					
SUB (opr)	Subtract	ACCA ← ACCA − M	IMM	A0	ii	2	2	—	—	↕	↕	↕
			DIR	B0	dd	2	3					
			EXT	C0	hh ll	3	4					
			IX2	D0	ee ff	3	5					
			IX1	E0	ff	2	4					
			IX	F0		1	3					
SWI	Software Interrupt	PC ← PC + 0001	INH	83		1	10	—	1	—	—	—
		(SP) ← PCL; SP ← SP − 0001										
		(SP) ← PCH; SP ← SP − 0001										
		(SP) ← X; SP ← SP − 0001										
		(SP) ← ACCA; SP ← SP − 0001										
		(SP) ← CC; SP ← SP − 0001										
		I bit ← 1										
		PCH ← $xFFC (vector										
		PCL ← $xFFD fetch)										
TAX	Transfer A to X	X ← ACCA	INH	97		1	2	—	—	—	—	—
TST (opr)	Test for Negative or Zero	M − 0	DIR	3D	dd	2	4	—	—	↕	↕	0
TSTA			INH(A)	4D		1	3					
TSTX			INH(X)	5D		1	3					
TST (opr)			IX1	6D	ff	2	5					
TST (opr)			IX	7D		1	4					
TXA	Transfer X to A	ACCA ← X	INH	9F		1	2	—	—	—	—	—
WAIT	Enable Interrupts, Halt CPU		INH	8F		1	2	—	0	—	—	—

3.4.7 Interrupts

Systems often require that normal processing be interrupted so that some external event may be serviced. The MC68HC705C8 may be interrupted by one of five different methods: any one of four maskable hardware interrupts (IRQ, SPI, SCI, or timer) and one nonmaskable software interrupt (SWI). Interrupts such as timer, SPI, and SCI have several flags which will cause the interrupt. Generally, interrupt flags are located in read-only status registers; their equivalent enable bits are located in associated control registers. The interrupt flags and enable bits are never contained in the same register. If the enable bit is a logic zero, it blocks the interrupt from occurring but does not inhibit the flag from being set. Reset clears all enable bits to preclude interrupts during the reset procedure.

The general sequence for clearing an interrupt is a software sequence of first accessing the status register while the interrupt flag is set, followed by a read or write of an associated register. When any of these interrupts occur and the enable bit is a logic one, normal processing is suspended at the end of the current instruction execution.

Figure 3-8 shows how interrupts fit into the normal flow of CPU instructions. Interrupts cause the processor registers to be saved on the stack and the interrupt mask (I bit) to be set to prevent additional interrupts. The appropriate interrupt vector then points to the starting address of the interrupt service

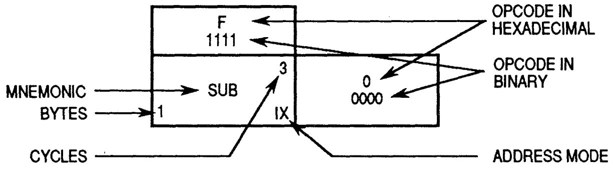
M68HC05 Instruction Set Opcode Map

	Bit Manipulation		Branch	Read/Modify/Write					Control		Register/Memory						
	BTB	BSC	REL	DIR	INH	INH	IX1	IX	INH	INH	IMM	DIR	EXT	IX2	IX1	IX	
HI LOW	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	HI LOW
0 0000	5 BRSET0 3 BTB	5 BSET0 2 BSC	3 BRA 2 REL	5 NEG 2 DIR	3 NEGA 1 INH	3 NEGX 1 INH	6 NEG 2 IX1	5 NEG 1 IX	9 RTI 1 INH		2 SUB 2 IMM	3 SUB 2 DIR	4 SUB 3 EXT	5 SUB 3 IX2	4 SUB 2 IX1	3 SUB 1 IX	0 0000
1 0001	5 BRCLR0 3 BTB	5 BCLR0 2 BSC	3 BRN 2 REL						6 RTS 1 INH		2 CMP 2 IMM	3 CMP 2 DIR	4 CMP 3 EXT	5 CMP 3 IX2	4 CMP 2 IX1	3 CMP 1 IX	1 0001
2 0010	5 BRSET1 3 BTB	5 BSET1 2 BSC	3 BHI 2 REL		11 MUL 1 INH						2 SBC 2 IMM	3 SBC 2 DIR	4 SBC 3 EXT	5 SBC 3 IX2	4 SBC 2 IX1	3 SBC 1 IX	2 0010
3 0011	5 BRCLR1 3 BTB	5 BCLR1 2 BSC	3 BLS 2 REL	5 COM 2 DIR	3 COMA 1 INH	3 COMX 1 INH	6 COM 2 IX1	5 COM 1 IX	10 SWI 1 INH		2 CPX 2 IMM	3 CPX 2 DIR	4 CPX 3 EXT	5 CPX 3 IX2	4 CPX 2 IX1	3 CPX 1 IX	3 0011
4 0100	5 BRSET2 3 BTB	5 BSET2 2 BSC	3 BCC 2 REL	5 LSR 2 DTR	3 LSRA 1 INH	3 LSRX 1 INH	6 LSR 2 IX1	5 LSR 1 IX			2 AND 2 IMM	3 AND 2 DIR	4 AND 3 EXT	5 AND 3 IX2	4 AND 2 IX1	3 AND 1 IX	4 0100
5 0101	5 BRCLR2 3 BTB	5 BCLR2 2 BSC	3 BCS 2 REL								2 BIT 2 IMM	3 BIT 2 DIR	4 BIT 3 EXT	5 BIT 3 IX2	4 BIT 2 IX1	3 BIT 1 IX	5 0101
6 0110	5 BRSET3 3 BTB	5 BSET3 2 BSC	3 BNE 2 REL	5 ROR 2 DIR	3 RORA 1 INH	3 RORX 1 INH	6 ROR 2 IX1	5 ROR 1 IX			2 LDA 2 IMM	3 LDA 2 DIR	4 LDA 3 EXT	5 LDA 3 IX2	4 LDA 2 IX1	3 LDA 1 IX	6 0110
7 0111	5 BRCLR3 3 BTB	5 BCLR3 2 BSC	3 BEQ 2 REL	5 ASR 2 DIR	3 ASRA 1 INH	3 ASRX 1 INH	6 ASR 2 IX1	5 ASR 1 IX		2 TAX 1 INH		4 STA 2 DIR	5 STA 3 EXT	6 STA 3 IX2	5 STA 2 IX1	4 STA 1 IX	7 0111
8 1000	5 BRSET4 3 BTB	5 BSET4 2 BSC	3 BHCC 2 REL	5 LSL 2 DIR	3 LSLA 1 INH	3 LSLX 1 INH	6 LSL 2 IX1	5 LSL 1 IX		2 CLC 1 INH	2 EOR 2 IMM	3 EOR 2 DIR	4 EOR 3 EXT	5 EOR 3 IX2	4 EOR 2 IX1	3 EOR 1 IX	8 1000
9 1001	5 BRCLR4 3 BTB	5 BCLR4 2 BSC	3 BHCS 2 REL	5 ROL 2 DIR	3 ROLA 1 INH	3 ROLX 1 INH	6 ROL 2 IX1	5 ROL 1 IX		2 SEC 1 INH	2 ADC 2 IMM	3 ADC 2 DIR	4 ADC 3 EXT	5 ADC 3 IX2	4 ADC 2 IX1	3 ADC 1 IX	9 1001
A 1010	5 BRSET5 3 BTB	5 BSET5 2 BSC	3 BPL 2 REL	5 DEC 2 DIR	3 DECA 1 INH	3 DECX 1 INH	6 DEC 2 IX1	5 DEC 1 IX		2 CLI 1 INH	2 ORA 2 IMM	3 ORA 2 DIR	4 ORA 3 EXT	5 ORA 3 IX2	4 ORA 2 IX1	3 ORA 1 IX	A 1010
B 1011	5 BRCLR5 3 BTB	5 BCLR5 2 BSC	3 BMI 2 REL							2 SEI 1 INH	2 ADD 2 IMM	3 ADD 2 DIR	4 ADD 3 EXT	5 ADD 3 IX2	4 ADD 2 IX1	3 ADD 1 IX	B 1011
C 1100	5 BRSET6 3 BTB	5 BSET6 2 BSC	3 BMC 2 REL	5 INC 2 DIR	3 INCA 1 INH	3 INCX 1 INH	6 INC 2 IX1	5 INC 1 IX		2 RSP 1 INH		2 JMP 2 DIR	3 JMP 3 EXT	4 JMP 3 IX2	3 JMP 2 IX1	2 JMP 1 IX	C 1100
D 1101	5 BRCLR6 3 BTB	5 BCLR6 2 BSC	3 BMS 2 REL	4 TST 2 DIR	3 TSTA 1 INH	3 TSTX 1 INH	5 TST 2 IX1	4 TST 1 IX		2 NOP 1 INH	6 BSR 2 REL	5 JSR 2 DIR	6 JSR 3 EXT	7 JSR 3 IX2	6 JSR 2 IX1	5 JSR 1 IX	D 1101
E 1110	5 BRSET7 3 BTB	5 BSET7 2 BSC	3 BIL 2 REL						2 STOP 1 INH		2 LDX 2 IMM	3 LDX 2 DIR	4 LDX 3 EXT	5 LDX 3 IX2	4 LDX 2 IX1	3 LDX 1 IX	E 1110
F 1111	5 BRCLR7 3 BTB	5 BCLR7 2 BSC	3 BIH 2 REL	5 CLR 2 DIR	3 CLRA 1 INH	3 CLRX 1 INH	6 CLR 2 IX1	5 CLR 1 IX	2 WAIT 1 INH	2 TXA 1 INH		4 STX 2 DIR	5 STX 3 EXT	6 STX 3 IX2	5 STX 2 IX1	4 STX 1 IX	F 1111

Abbreviations for Address Modes

INH	Inherent
A	Accumulator
X	Index Register
IMM	Immediate
DIR	Direct
EXT	Extended
REL	Relative
BSC	Bit Set/Clear
BTB	Bit Test and Branch
IX	Indexed (No Offset)
IX1	Indexed, 1 Byte (8-Bit) Offset
IX2	Indexed, 2 Byte (16-Bit) Offset

Legend



routine (refer to Figure 3-9 and Table 3-6 for vector location). Upon completion of the interrupt service routine, the RTI instruction (which is normally the last instruction of the routine) causes the register contents to be recovered from the stack followed by a return to normal processing.

NOTE

The interrupt mask bit (I bit) will be cleared if, and only if, the corresponding bit stored in the stack is zero.

Table 3-6. Vector Address for Interrupts and Reset

Register	Flag Name	Interrupts	CPU Interrupt	Vector Address
N/A	N/A	Reset	$\overline{\text{RESET}}$	\$1FFE–\$1FFF
N/A	N/A	Software	SWI	\$1FFC–\$1FFD
N/A	N/A	External Interrupt	$\overline{\text{IRQ}}$	\$1FFA–\$1FFB
Timer Status	ICF OFC TOF	Input Capture Output Compare Timer Overflow	TIMER	\$1FF8–\$1FF9
SCI Status	TDRE TC RDRF IDLE OR	Transmit Buffer Empty Transmit Complete Receiver Buffer Full Idle Line Detect Overrun	SCI	\$1FF6–\$1FF7
SPI Status	SPIF MODF	Transfer Complete Mode Fault	SPI	\$1FF4–\$1FF5

Reset and interrupt operations are often discussed together because they share the common concept of vector fetching to force a new starting point for further CPU operation. Unlike interrupts, there is no intention to ever return to whatever the CPU was doing before a reset occurred.

A low on the $\overline{\text{RESET}}$ input pin causes the program to vector to its starting address specified by the contents of memory location \$1FFE and \$1FFF. The I bit in the condition code register is also set. Much of the MCU is configured (forced) to a known state during reset.

3.4.7.1 SOFTWARE INTERRUPT (SWI). The software interrupt is an executable instruction. The action of the SWI instruction is similar to the hardware interrupts. The SWI is executed regardless of the state of the interrupt mask (I bit) in the condition code register. The interrupt service routine address is specified by the contents of memory location \$1FFC and \$1FFD.

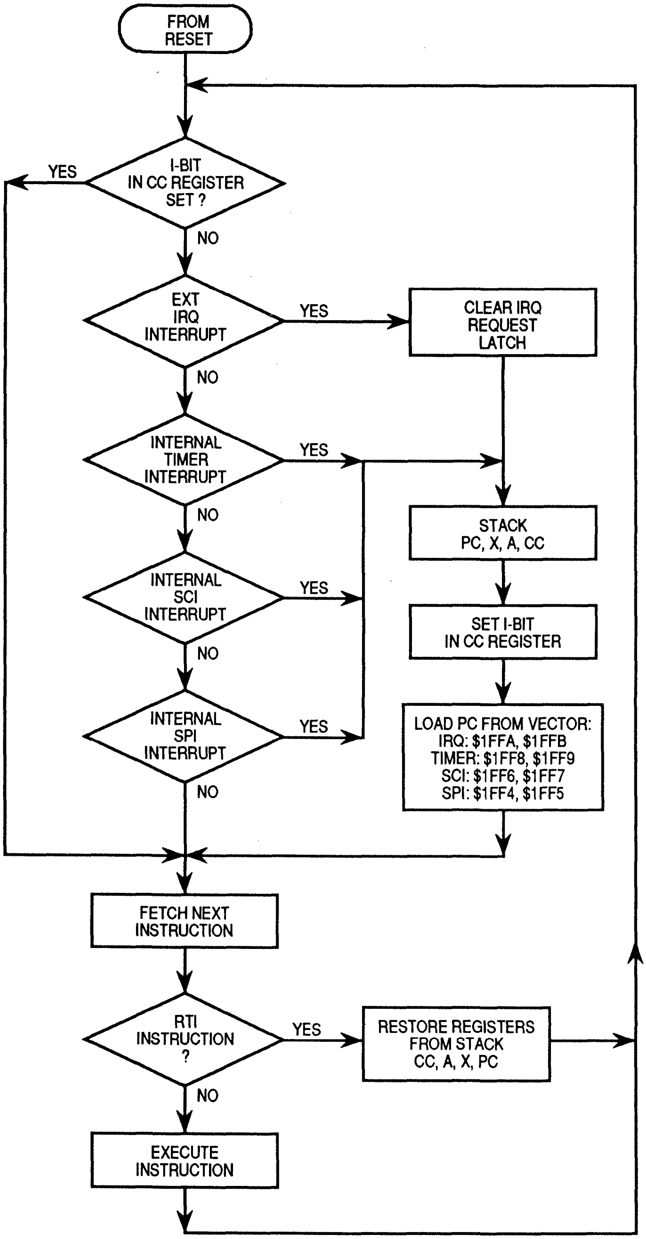
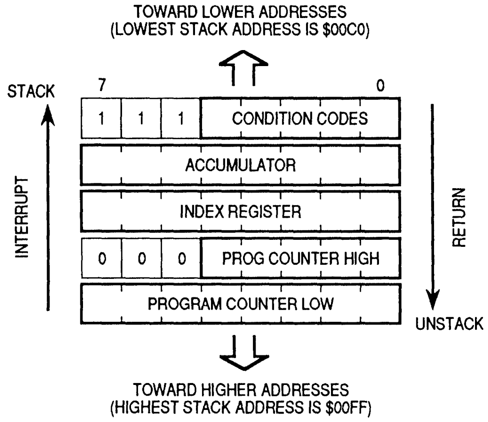


Figure 3-8. Hardware Interrupt Flowchart



NOTE: When an interrupt occurs, CPU registers are saved on the stack in the order PCL, PCH, X, A, CC. On a return from interrupt registers are recovered from the stack in reverse order.

Figure 3-9. Interrupt Stacking Order

3.4.7.2 EXTERNAL INTERRUPT. If the interrupt mask (I bit) of the condition code register has been cleared and the external interrupt pin ($\overline{\text{IRQ}}$) has gone low, then the external interrupt is recognized. When the interrupt is recognized, the current state of the CPU is pushed onto the stack and the I bit is set. This masks further interrupts until the present one is serviced. The interrupt service routine address is specified by the contents of memory location $1FFA and $1FFB.

The MC68HC705C8 MCU $\overline{\text{IRQ}}$ pin sensitivity is software programmable. Either negative edge- and level-sensitive triggering or negative edge-sensitive triggering are available. The MC68HC705C8 MCU uses the option register residing at location $1FDF to control the $\overline{\text{IRQ}}$ pin sensitivity.

3.4.7.3 TIMER INTERRUPT. There are three different interrupt flags that will cause a timer interrupt whenever they are set and enabled. These three interrupt flags are found in the three MSBs of the timer status register (TSR, location $13), and all three will vector to the same interrupt service routine ($1FF8–$1FF9).

All interrupt flags have corresponding enable bits (ICIE, OCIE, and TOIE) in the timer control register (TCR, location $12). Reset clears all enable bits, thus preventing an interrupt from occurring during the reset time period. The actual processor interrupt is generated only if the I bit in the condition code register is also cleared. The general sequence for clearing an interrupt is a software sequence of accessing the status register while the flag is set, followed by a read or write of the associated control register.

3.4.7.4 SERIAL COMMUNICATIONS INTERFACE (SCI) INTERRUPT. An interrupt in the SCI occurs when one of the interrupt flag bits in the serial communications status register is set, provided the I bit in the condition code register is clear and the enable bit in the serial communication control register 2 (location $0F) is enabled. Software in the serial interrupt service routine must determine the priority and cause of the SCI interrupt by examining the interrupt flags and the status bits located in the serial communications status register (location $10) The general sequence for clearing an interrupt is a software sequence of accessing the status register while the flag is set, followed by a read or write of the associated control register.

3.4.7.5 SERIAL PERIPHERAL INTERFACE (SPI) INTERRUPT. An interrupt in the SPI occurs when one of the interrupt flag bits in the serial peripheral status register (location $0B) is set, provided the I bit in the condition code register is clear and the enable bit in the serial peripheral control register (location $0A) is enabled. The general sequence for clearing an interrupt is a software sequence of accessing the status register while the flag is set, followed by a read or write of the associated control register.

3.5 MICROCONTROLLER INPUT/OUTPUT

Since inputs to and outputs from the MCU are usually digital (0 to +5 Vdc at low power), interface logic is often needed to couple the MCU to external devices. Interface logic can operate in parallel or serial form.

Parallel interfaces allow I/O data transfer eight bits at a time, to parallel ports on the MCU. Serial interfaces transfer I/O data one bit at a time through a serial communications interface (SCI) or serial peripheral interface (SPI) that are parts of the MCU.

Data transfers between the MCU and external logic are controlled by the MCU.

NOTE

Tie all unused inputs and I/O ports to an appropriate logic level, either V_{DD} or V_{SS}.

3.5.1 Parallel I/O

The MC68HC705C8 MCU contains 31 general-purpose parallel I/O pins arranged in four ports. Ports A, B, and C are 8-bit ports in which the direction of each pin is programmable by software-accessible registers. Each 8-bit port has an associated 8-bit data direction register (DDR) as shown in Figures 3-10, 3-11, and 3-12.

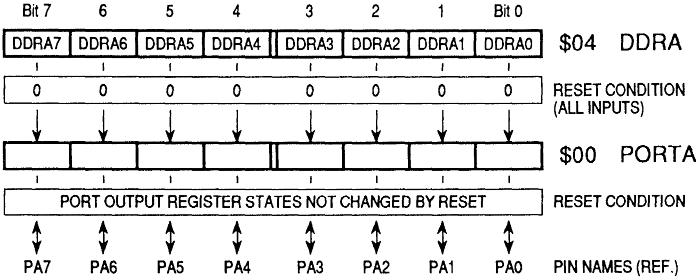


Figure 3-10. Port A and Data Direction A Registers

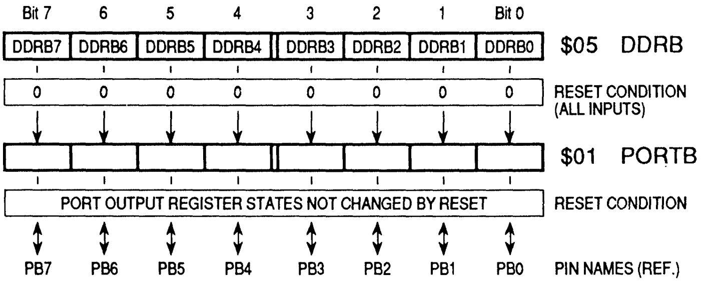


Figure 3-11. Port B and Data Direction B Registers

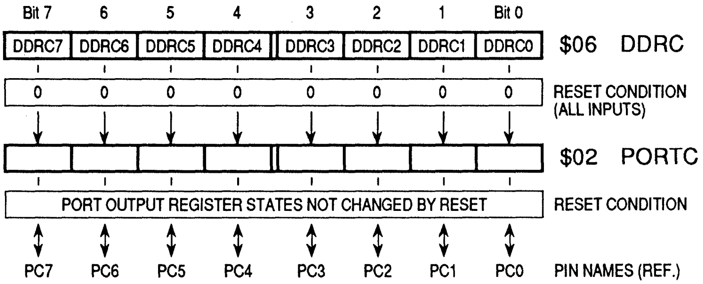
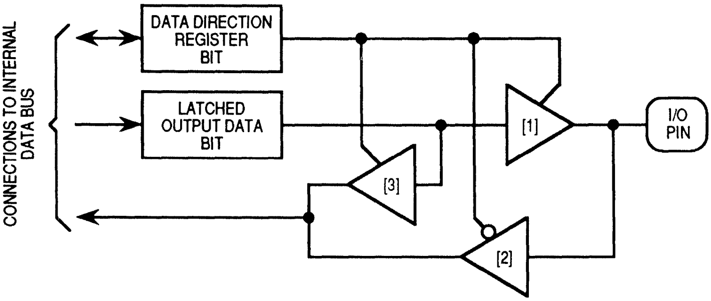


Figure 3-12. Port C and Data Direction C Registers

Any port A, B, or C pin is configured as an output if its corresponding DDR bit is set to a logic one. A pin is configured as an input if its corresponding DDR bit is cleared to a logic zero. At power-on or reset, all DDRs are cleared, which configure all port A, B, and C pins as inputs. The DDRs are capable of being written to or being read by the processor. Refer to Figure 3-13 and Table 3-7. When a port pin is configured as an output, a read of the data register actually reads the value of the output data latch and not the I/O pin.



[1] - Output Buffer, enables latched output to drive pin when DDR bit is 1 (output)

[2] - Input Buffer, enabled when DDR bit is 0 (Input).

[3] - Input Buffer, enabled when DDR bit is 1 (Output).

Figure 3-13. Parallel Port I/O Circuitry

Table 3-7. I/O Pin Functions

R/$\overline{W}$*	DDR	I/O Pin Function
0	0	The I/O pin is in input mode. Data is written into the output data latch.
0	1	Data is written into the output data latch and output to the I/O pin.
1	0	The state of the I/O pin is read.
1	1	The I/O pin is in output mode. The output data latch is read.

*R/$\overline{W}$ is an internal signal.

3.5.2 Serial I/O

Port D (see Figure 3-14) is a 7-bit fixed-direction input port. The SPI and SCI systems take control of port D pins when these systems are enabled. During power-on reset or external reset, all seven pins (PD5–PD0, PD7) are configured as input ports because all special-function output drivers are disabled. For example, with the SCI system enabled (RE = TE = 1), PD0 and PD1 inputs will read zero. With the SPI system disabled (SPE = 0), PD5–PD2 will read the state of the pin at the time of the read operation.

The SCI function uses two of the pins (PD1–PD0) for its receive data input (RDI) and transmit data output (TDO); the SPI function uses four of the pins (PD5–PD2) for its serial data input/output (MISO, MOSI), system clock (SCK), and slave select (SS), respectively.

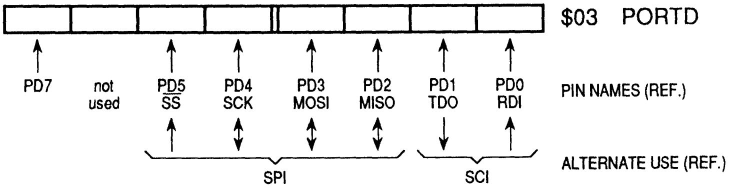


Figure 3-14. Port D Fixed Input Port

3.6 SERIAL COMMUNICATIONS INTERFACE (SCI)

SCI is one of two independent serial I/O subsystems in the MC68HC705C8. The other serial I/O system (called SPI) provides for high-speed synchronous serial communication to peripherals or other MCUs. The SCI is a full-duplex UART-type asynchronous system that can be used for communication between the MCU and a CRT terminal or a personal computer, or several widely distributed MCUs can use their SCI subsystems to form a serial communications network.

The SCI uses standard nonreturn-to-zero (NRZ) format (one start bit, eight or nine data bits, and a stop bit). The most common data format is eight bits. An on-chip baud rate generator derives standard baud rate frequencies from the MCU oscillator. The SCI transmitter and receiver are functionally independent but use the same data format and baud rate. In this applications guide, "baud rate" and "bit rate" are used synonymously.

SCI Features:

- Two-Wire Serial Interface
- Standard NRZ (mark/space) Format
- Full-Duplex Operation (independent transmit and receive)
- Software Programmable for One of 32 Different Baud Rates
- Software-Selectable Word Length (8- or 9-bit words)
- Separate Transmitter and Receiver Enable Bits
- Communication may be Interrupt Driven

Receiver:

- Receiver Data Register Full Flag
- Error Detect Flags — Framing, Noise, Overrun
- Idle-Line Detect Flag
- Receiver Wakeup Function (idle or address bit)

Transmitter:

- Transmit Data Register Empty Flag
- Transmit Complete Flag (for modem control)
- Break Send

3.6.1 SCI Transmitter

The SCI transmitter block diagram is shown in Figure 3-15. The heart of the transmitter is the transmit serial shift register near the top of the figure. Usually, this shift register obtains its data from the write-only transmit buffer. Data is transferred into the transmit buffer when software writes to the SCI data register (SCDAT). Whenever data is transferred into the shifter from the transmit buffer, a zero is loaded into the LSB of the shifter to act as start bit, and a logic one is loaded into the last bit position to act as a stop bit. In the case of a preamble, the shifter is loaded with all ones, including the bit position usually holding the logic zero start bit. A preamble is loaded each time the transmit enable bit is written from zero to one. In the case of a send

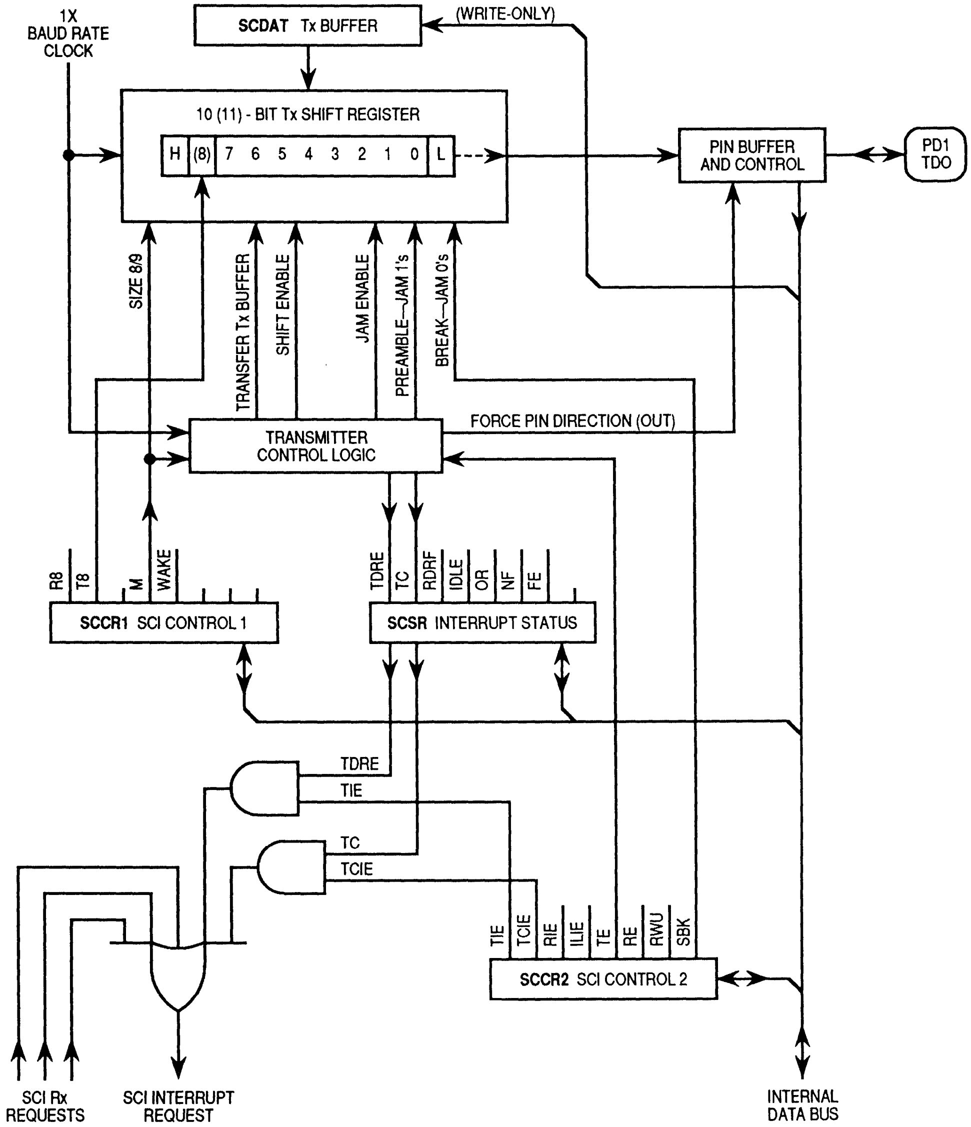


Figure 3-15. SCI Transmitter Block Diagram

break command, the shifter is loaded with all zeros, including the last bit position usually holding the logic one stop bit.

The T8 bit in SCI control register 1 (SCCR1) acts like an extra high-order bit (ninth bit) of the transmit buffer register. This ninth bit is only used if the M bit in SCCR1 is set, selecting the 9-bit data character format. The M bit also controls the length of idle and break characters.

The status flag and interrupt generation logic are shown in Figure 3-15. The transmit data register empty (TDRE) and transmit complete (TC) status flags in the SCI status register (SCSR) are automatically set by the transmitter logic. These two bits can be read at any time by software. The transmit interrupt enable (TIE) and transmit complete interrupt enable (TCIE) control bits enable the TDRE and TC flags, respectively, to generate SCI interrupt requests.

3.6.2 SCI Receiver

The receiver block diagram is shown in Figure 3-16. SCI received data comes in on the RDI pin, is buffered, and drives the data recovery block. The data recovery block is actually a high-speed shifter operating at 16 times the bit rate; the main receive serial shifter operates at one times the bit rate. This higher speed sample rate allows the start-bit leading edge to be located more accurately than a 1× clock would allow. The high-speed clock also allows several samples to be taken within a bit time so logic can make an intelligent decision about the logic sense of a bit (even in the presence of noise). The data recovery block provides the bit level to the main receiver shift register and also provides a noise flag status indication.

The heart of the receiver is the receive serial shift register. This register is enabled by the receive enable (RE) bit in the SCI control register 2 (SCCR2). The M bit from the SCCR1 register determines whether the shifter will be 10 or 11 bits. After detecting the stop bit of a character, the received data is transferred from the shifter to the SCDAT, and the receive data register full (RDRF) status flag is set. When a character is ready to be transferred to the receive buffer but the previous character has not yet been read, an overrun condition occurs. In the overrun condition, data is not transferred, and the overrun (OR) status flag is set to indicate the error.

There are three receiver-related interrupt sources in the SCI. These flags can be polled by software or, when enabled, cause an SCI interrupt request. The receive interrupt enable (RIE) control bit enables the RDRF and OR status

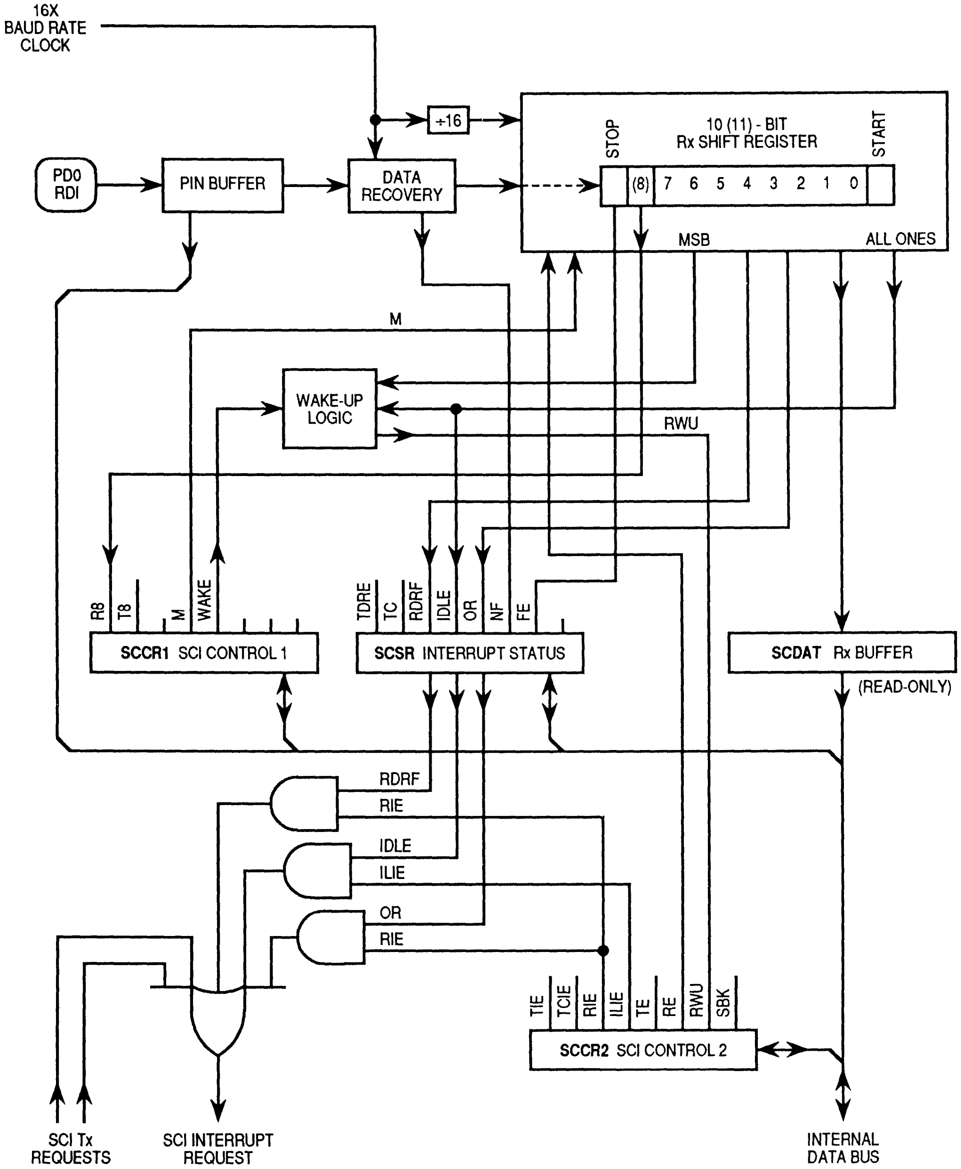


Figure 3-16. SCI Receiver Block Diagram

flags to generate hardware interrupt requests. The idle line interrupt enable (ILIE) control bit allows the IDLE status flag to generate interrupt requests.

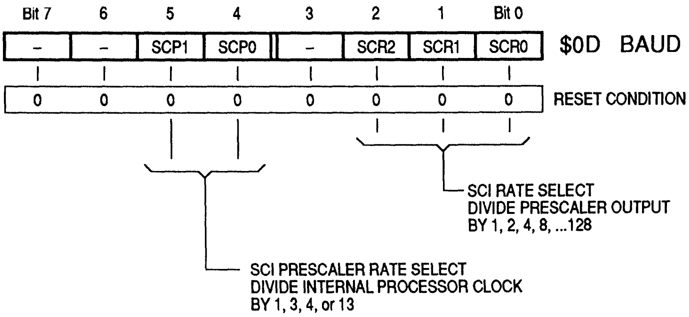
3.6.3 Registers

The SCI system includes five registers (BAUD, SCCR1, SCCR2, SCSR, and SCDAT) and two external pins (TDO and RDI). When the SCI receiver and/or transmitter is enabled, the SCI logic takes control of the pin buffers for the associated port D pin(s). When the SCI is disabled, the TDO and RDI pins act as general-purpose inputs.

The main function of each of these registers will be discussed. Normally, the SCCR1, SCCR2, and BAUD registers would be written once to initialize and then not used again. An example of the software/programming procedure is shown later in this section.

3.6.3.1 BAUD RATE REGISTER (BAUD). The BAUD register (see Figure 3-17) is used to select the baud rate for the SCI system. Both the transmitter and receiver use the same data format and baud rate, which is derived from the MCU bus rate clock. The SCP1–SCP0 bits function as a prescaler for the SCR2–SCR0 bits. Together, these five bits provide multiple baud rate combinations for a given crystal frequency.

The diagram shown in Figure 3-18 and Tables 3-8 and 3-9 illustrate the divider chain used to obtain the baud rate clock (transmit clock). For example, using a 4-MHz crystal, the internal processor clock is 2 MHz.



Bit 7	6	5	4	3	2	1	Bit 0	
–	–	SCP1	SCP0	–	SCR2	SCR1	SCR0	$0D BAUD
0	0	0	0	0	0	0	0	RESET CONDITION

Figure 3-17. Baud Rate Register

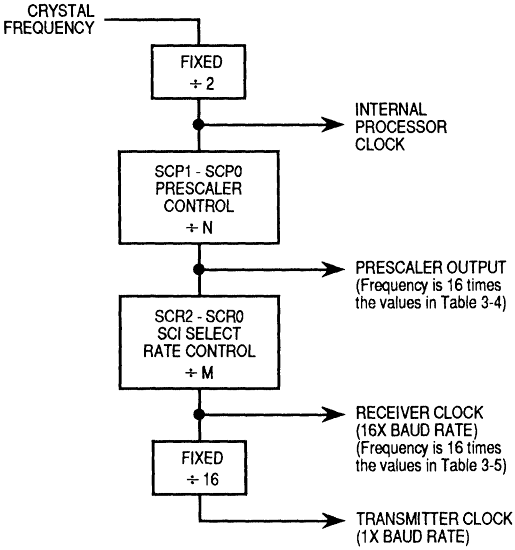


Figure 3-18. Rate Generator Division

Table 3-8. Prescaler Baud Rate Frequency Output

SCP Bit		Clock* Divided By	Crystal Frequency MHz				
1	0		4.194304	4.0	2.4576	2.0	1.8432
0	0	1	131.072 kHz	125.000 kHz	76.80 kHz	62.60 kHz	57.60 kHz
0	1	3	43.691 kHz	41.666 kHz	25.60 kHz	20.833 kHz	19.20 kHz
1	0	4	32.768 kHz	31.250 kHz	19.20 kHz	15.625 kHz	14.40 kHz
1	1	13	10.082 kHz	9600 Hz	5.907 kHz	4800 Hz	4430 Hz

*The clock in the "Clock Divided By" column is the internal processor clock.

NOTE: The divided frequencies shown in Table 3-8 represent baud rates which are the highest transmit baud rate (Tx) that can be obtained by a specific crystal frequency and only using the prescaler division. Lower baud rates may be obtained by providing a further division using the SCI rate select bits shown below for some representative prescaler outputs.

The SCP1–SCP0 bits in the baud rate register set the division factor (N in Figure 3-18) for the baud rate divider. Reset clears these bits, setting the prescaler to divide-by-one.

The SCR2, SCR1, and SCR0 bits are used to set the division factor (M in Figure 3-18) for the baud rate divider. Reset does not affect these bits.

Example:

From Table 3-8, find the crystal frequency used (in this case, 4 MHz). Next, find 9600 or a binary multiple of 9600. In this example, you would select

Table 3-9. Transmit Baud Rate Output

SCR Bits			Divided By	Representative Highest Prescaler Baud Rate Output				
2	1	0		131.072 kHz	32.768 kHz	76.80 kHz	19.20 kHz	9600 Hz
0	0	0	1	131.072 kHz	32.768 kHz	76.80 kHz	19.20 kHz	9600 Hz
0	0	1	2	65.536 kHz	16.384 kHz	38.40 kHz	9600 Hz	4800 Hz
0	1	0	4	32.768 kHz	8.192 kHz	19.20 kHz	4800 Hz	2400 Hz
0	1	1	8	16.384 kHz	4.096 kHz	9600 Hz	2400 Hz	1200 Hz
1	0	0	16	8.192 kHz	2.048 kHz	4800 Hz	1200 Hz	600 Hz
1	0	1	32	4.096 kHz	1.024 kHz	2400 Hz	600 Hz	300 Hz
1	1	0	64	2.048 kHz	512 Hz	1200 Hz	300 Hz	150 Hz
1	1	1	128	1.024 kHz	256 Hz	600 Hz	150 Hz	75 Hz

NOTE: Table 3-9 illustrates how the SCI select bits can be used to provide lower transmitter baud rate by further dividing the prescaler output frequency. The five examples are only representative samples. In all cases, the baud rates shown are transmit baud rates (transmit clock), and the receive clock is 16 times higher in frequency than the actual baud rate.

the bottom row which corresponds to SCP1:SCP0 = 1:1 (divide-by-thirteen). Next, find the column in Table 3-5 that corresponds to 9600 Hz. Find the desired baud rate in this column. In this example, you would select the top row, which corresponds to SCR2:SCR1:SCR0 = 0:0:0 (divide-by-one).

3.6.3.2 SERIAL COMMUNICATIONS CONTROL REGISTER ONE (SCCR1).

The serial communications control register one (SCCR1) shown in Figure 3-19 includes three bits associated with the optional 9-bit data format. The WAKE bit is used to select one of two methods of receiver wakeup. Normal setup for bit M is 0 for 8-bit words. The other register bits are not used in most systems. In a typical system, this register would be written to $00 during initialization.

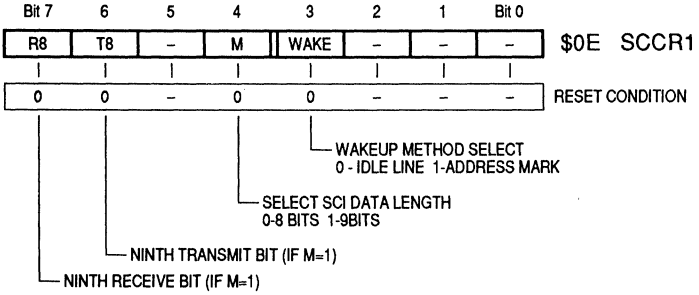


Figure 3-19. Serial Communications Control Register One

3.6.3.3 SERIAL COMMUNICATIONS CONTROL REGISTER TWO (SCCR2). The serial communications control register two (SCCR2) shown in Figure 3-20 is the main control register for the SCI subsystem. This register can enable/ disable the transmitter or receiver, enable the system interrupts, and provide the wakeup enable bit and a "send break code" bit. The TIE, TCIE, RIE, and ILIE bits are local interrupt enable controls, which determine whether SCI status flags will be polled or generate hardware interrupt requests.

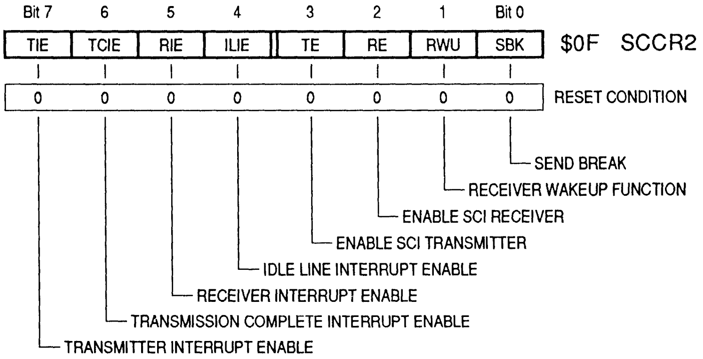


Figure 3-20. Serial Communications Control Register Two

In a typical system:

TE and RE would be written to one to enable the transmitter and receiver subsystems.

ILIE, RWU, and SBK would seldom be used and would be written to zero.

If interrupts were not being used, TIE, TCIE, and RIE would be written to zero. If interrupts were used, these three bits would be written to one.

For example, in a system which does not use interrupts, SCCR2 would be loaded with $0C during initialization.

3.6.3.4 SERIAL COMMUNICATIONS STATUS REGISTER (SCSR). The SCI status register (SCSR) in Figure 3-21 contains two transmitter status flags and five receiver related status flags. The TDRE and RDRF bits are always used. The TC and IDLE bits are not commonly used.

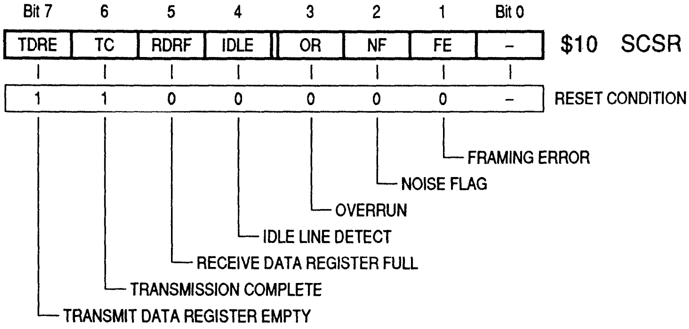


Figure 3-21. Serial Comunications Status Register

The OR, NF, and FE bits should be monitored and may or may not be used, depending on the type of SCI system. For errors to be corrected, both the transmitting and receiving device must have a common method of handling errors.

There are two major types of communication links associated with the SCI. An example of a direct connection would be an MCU connected to a personal computer. In this direct connection link OR, NF, and FE errors are very unlikely and are typically ignored. The second type of link involves two remote devices where each is connected to a modem. In this type of link, errors are more likely and both computers would typically use a protocol that permits retransmission when an error is detected.

3.6.3.5 SERIAL COMMUNICATIONS DATA REGISTER (SCDAT).

The SCI SCDAT data register (see Figure 3-22) has two functions: it is the transmit data register when written to and the receive data register when read. Both the transmitter and receiver are double buffered (see Figure 3-23), so back-to-back characters can be handled easily even if the CPU is delayed in responding to the completion of an individual character.

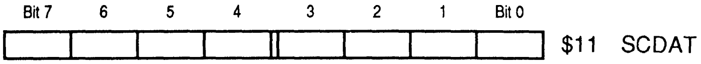


Figure 3-22. Serial Communications Data Register

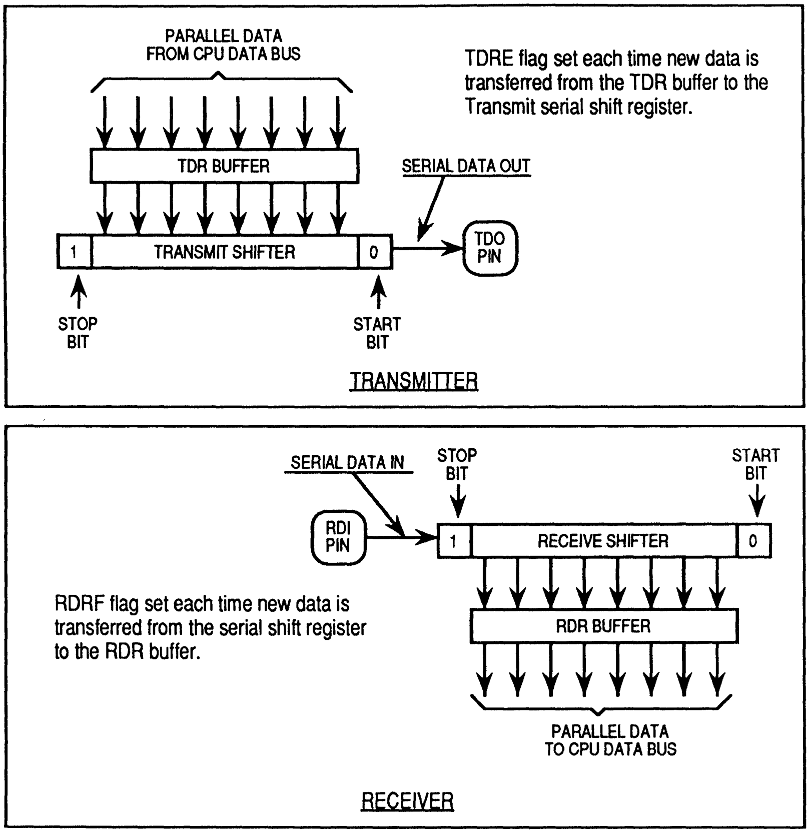

PARALLEL DATA FROM CPU DATA BUS
TDRE flag set each time new data is transferred from the TDR buffer to the Transmit serial shift register.
TDR BUFFER
SERIAL DATA OUT
1
TRANSMIT SHIFTER
0
TDO PIN
STOP BIT
START BIT
TRANSMITTER
SERIAL DATA IN
STOP BIT
START BIT
RDI PIN
1
RECEIVE SHIFTER
0
RDRF flag set each time new data is transferred from the serial shift register to the RDR buffer.
RDR BUFFER
PARALLEL DATA TO CPU DATA BUS
RECEIVER

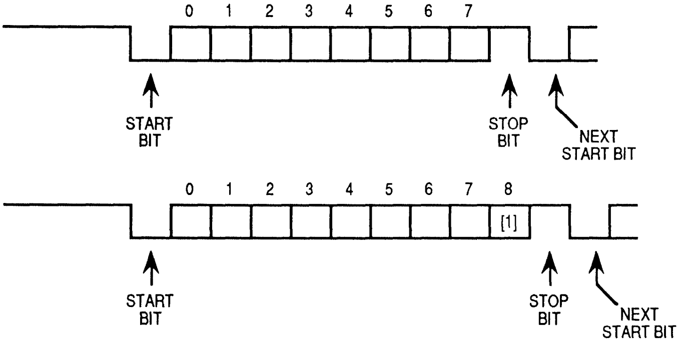
Figure 3-23. Double Buffering

3.6.4 Data Formats

The standard NRZ data formats used for communications are shown in Figure 3-24. The upper portion of this figure shows the normal 8-bit data format; the lower portion of the figure shows the 9-bit data format. The 9-bit data format is selected by setting the M control bit in SCCR1 to 1.

The basic characteristics of the NRZ format are as follows:

1) A high level indicates a logic one and a low level, a logic zero.
2) The idle line is high prior to message transmission/reception.
3) A start bit (logic zero) is transmitted/received as the first bit of data in a character.
4) Data is transmitted/received LSB first.
5) The last bit in a character (bit 10 or 11) is a high (stop bit).
6) A break is a low (logic zero) for 10 or 11 bit times.



[1] - Control bit "M" selects optional ninth data bit.

Figure 3-24. Data Formats

3.6.5 Hardware Procedures

Some simple hardware setup is required. A universal standard RS232 cable is used to interconnect the SCI to a CRT terminal or the PC. The user would usually have to provide an external level shifter buffer (MC145406) to convert the RS232 (typically ±12 volts) to the 0–5 volt logic levels used by the MC68HC705C8.

3.6.6 Software Procedures

The following paragraphs and flowcharts discuss software procedures. These flowcharts illustrate how straightforward normal SCI operations are.

3.6.6.1 INITIALIZATION PROCEDURE. The following list reflects the initialization procedure.

1) Write to BAUD register (SCP1–SCP0, SCR2–SCR0) to set baud rate.

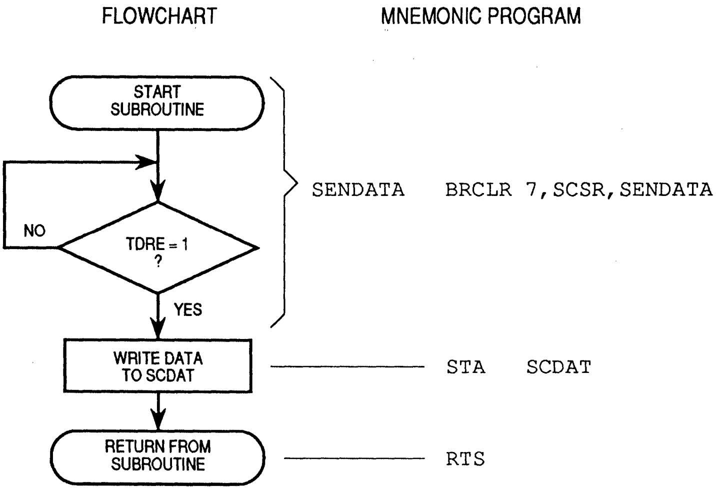
2) Write to SCCR1 (R8, T8, M, WAKE) to set character length and choose wakeup method.

3) Write to SCCR2 (TIE, TCIE, RIE, ILIE, TE, RE, RWU, SBK) to enable desired interrupt sources. To turn on the transmitter and receiver, RWU and SBK would be written to zero during initialization.

The following is a reference list of interrupt enable control bits versus the interrupt source(s) they enable:

Enable	Flags	Interrupt Source Names
TIE	TDRE	Transmit data register empty
TCIE	TC	Transmit complete
RIE	RDRF, OR	Receive data register full, overrun
ILIE	IDLE	Idle line detect

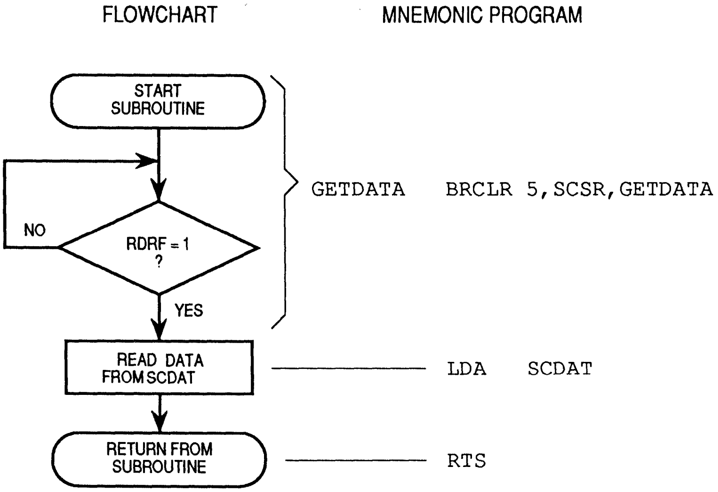
3.6.6.2 NORMAL TRANSMIT OPERATION. Refer to Figure 3-25, a flowchart of the normal transmit operation.



```
SENDATA    BRCLR 7,SCSR,SENDATA
           STA    SCDAT
           RTS
```

Figure 3-25. SCI Normal Transmit Operation Flowchart

3.6.6.3 NORMAL RECEIVE OPERATION. Refer to Figure 3-26, a flowchart of the normal receive operation.



```
GETDATA    BRCLR 5,SCSR,GETDATA
           LDA    SCDAT
           RTS
```

Figure 3-26. SCI Normal Receive Operation Flowchart

3.6.7 SCI Application Example

Figure 3-27 is an example software program for communication between the SCI of the MCU and a dumb terminal. The MCU will receive (read) an ASCII character that was sent by the dumb terminal. The MCU will then translate the 8-bit binary character representing the ASCII character into two ASCII characters.

When this translation is completed, the MCU will transmit a <CR>, line feed, a $ sign and the two characters that represent the original hexadecimal equivalent of the received character back to the terminal. The program then waits for another character.

In practice, the following would occur:

> You type a number/character on the keyboard. It goes from the terminal to the MCU over the SCI receiver. Use the example of the letter "A".
>
> The program translates "A" to "4" and "1", then sends CR, line feed, $, 4, and 1, to the SCI transmitter.
>
> When the transmission is complete, the program goes back to the top for another keyboard number/character to be sent over the SCI receiver.

Table 3-10 is a chart of the ASCII-hexadecimal code conversion.

Table 3-10. ASCII-Hexadecimal Code Conversion

ASCII CHARACTER SET (7-BIT CODE)								
MS Dig. LS Dig.	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	(a	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	'	<	L	v	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	\	n	~
F	SI	US	/	/	O	—	o	DEL

```
                          ********************************************
                          * Simple 68HC05 SCI Program Example        *
                          ********************************************

000d                      BRATE    EQU   $0D       -,-,SCP1,SCP0;-,SCR2,SCR1,SCR0
000e                      SCCR1    EQU   $0E       R8,T8,-,M;WAKE,-,-,-
000f                      SCCR2    EQU   $0F       TIE,TCIE,RIE,ILIE;TE,RE,RWU,SBK
0011                      SCDAT    EQU   $11       Read - RDR; Write - TDR
0010                      SCSR     EQU   $10       TDRE,TC,RDRF,IDLE;OR,NF,FE,-

00a0                      TEMP     EQU   $A0       One byte temp storage location
00a1                      TEMPHI   EQU   $A1       Upper byte changed to ASCII
00a2                      TEMPLO   EQU   $A2       Lower byte changed to ASCII

0500                               ORG   $500      Program will start at $0500

0500 a6 30              INITIAL  LDA   #%00110000  Begin initialization
0502 b7 0d                         STA   BRATE        Baud rate to 4800 @2MHz Xtal
0504 a6 00                         LDA   #%00000000  Set up SCCR1
0506 b7 0e                         STA   SCCR1        Store in SCCR1 register
0508 a6 0c                         LDA   #%00001100  Set up SCCR2
050a b7 0f                         STA   SCCR2        Store in SCCR2 register
050c cd 05 43  START    JSR   GETDATA   Checks for receive data
050f b7 a0                         STA   TEMP      Store received ASCII data in temp
0511 a4 0f                         AND   #$0F      Convert LSB of ASCII char to hex
0513 aa 30                         ORA   #$30      $3(LSB) = "LSB"
0515 a1 39                         CMP   #$39      3A-3F need to change to 41-46
0517 23 02                         BLS   ARN1      Branch if 30-39 OK
0519 ab 07                         ADD   #7        Add offset
051b b7 a2        ARN1     STA   TEMPLO    Store LSB of hex in TEMPLO
051d b6 a0                         LDA   TEMP      Read the original ASCII data
051f 44                            LSRA            Shift right 4 bits
0520 44                            LSRA
0521 44                            LSRA
0522 44                            LSRA
0523 aa 30                         ORA   #$30      ASCII for N is $3N (N=0-9)
0525 a1 39                         CMP   #$39      3A-3F need to change to 41-46
0527 23 02                         BLS   ARN2      Branch if 30-39
0529 ab 07                         ADD   #7        Add offset
052b b7 a1        ARN2     STA   TEMPHI    MS nibble of hex to TEMPHI
052d a6 0d                         LDA   #$0D      Load hex value for "<CR>"
052f ad 18                         BSR   SENDATA   Carriage return
0531 a6 0a                         LDA   #$0A      Load hex value for "<LF>"
0533 ad 14                         BSR   SENDATA   Line feed
0535 a6 24                         LDA   #'$       Load hex value for "$"
0537 ad 10                         BSR   SENDATA   Print dollar sign
0539 b6 a1                         LDA   TEMPHI    Get high half of hex value
053b ad 0c                         BSR   SENDATA   Print
053d b6 a2                         LDA   TEMPLO    Get low half of hex value
053f ad 08                         BSR   SENDATA   Print
0541 20 c9                         BRA   START     Branch back to start
```

Figure 3-27. SCI Application Example Program (Sheet 1 of 2)

```
                  *** Get an SCI character, return w/ it in A
0543 0b 10 fd GETDATA  BRCLR 5,SCSR,GETDATA RDRF = 1 ?
0546 b6 11             LDA   SCDAT           OK, get
0548 81                RTS                   ** Return from GETDATA **

                  *** Send an SCI character, call sub w/ it in A
0549 0f 10 fd SENDATA  BRCLR 7,SCSR,SENDATA TDRE = 1 ?
054c b7 11             STA   SCDAT           OK, send
054e 81                RTS                   ** Return from SENDATA **
```

Figure 3-27. SCI Application Example Program (Sheet 2 of 2)

3.7 SYNCHRONOUS SERIAL PERIPHERAL INTERFACE (SPI)

The SPI subsystem included in the MC68HC705C8 allows the MCU to communicate with peripheral devices. Peripheral devices can be as simple as an ordinary TTL shift register or as complex as a complete subsystem such as an LCD display driver or an A/D converter subsystem. The SPI system is flexible enough to interface directly with numerous standard product peripherals from several manufacturers.

SPI is an added feature for those applications that require more inputs and outputs than there are parallel I/O pins on the MCU. SPI offers a very easy way to expand the I/O function while using a minimum number of MCU pins. The SPI block diagram is shown in Figure 3-28.

SPI features are as follows:

- Full-Duplex, Three-Wire Synchronous Transfers
- Master or Slave Operation
- 1.05 MHz (maximum) Master Bit Frequency
- 2.1 MHz (maximum) Slave Bit Frequency
- Four Programmable Master Bit Rates
- Programmable Clock Polarity and Phase
- End of Transmission Interrupt Flag
- Write-Collision Flag Protection

An SPI subsystem can operate under software control in either complex or simple system configurations:

- One Master MCU and Several Slave MCUs
- Several MCUs Interconnected in a Multimaster System
- One Master MCU and One or More Slave Peripherals

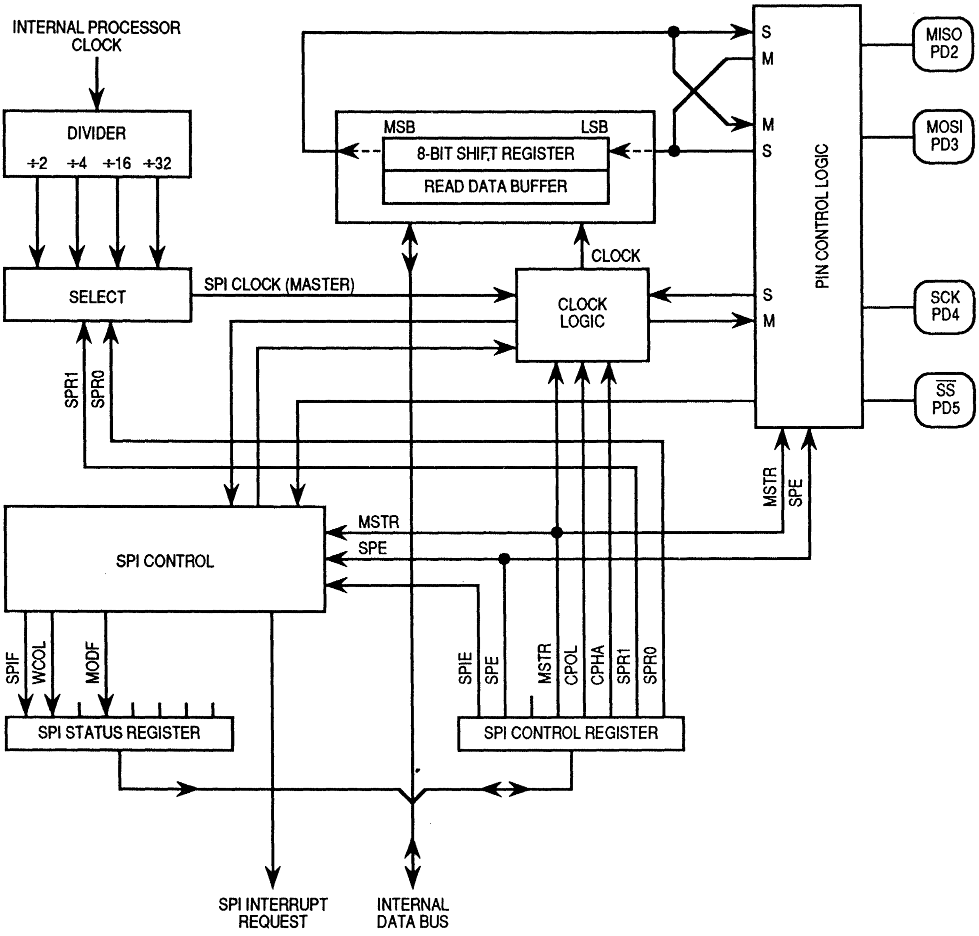


Figure 3-28. SPI Block Diagram

The majority of all applications use one MCU device as the master. This master initiates and controls the transfer of data to/from one or more slave peripheral devices that receive/supply the data being transferred. Slaves can read data from or transfer data to the master only after the master instructs an action to occur. This system configuration will be discussed in this applications guide.

3.7.1 Data Movement

There is no need to specify the direction of data movement for each transfer because the master simultaneously transmits and receives serial data on separate pins every transfer. When an SPI transfer occurs, an 8-bit character is shifted out on one data pin while a different 8-bit character is simultaneously shifted in on a second data pin (see Figure 3-29). Another way to think of this is that an 8-bit shift register in the master and another in the slave are connected as a circular 16-bit shift register. When a transfer occurs, this distributed shift register is shifted eight bit positions so the characters in the master and slave are effectively exchanged.

Many simple slave devices are designed to only receive data from a master or only supply data to a master. For example, a serial-to-parallel shift register can act as an 8-bit output port. An MCU configured as a master SPI device would initiate a transfer to send an 8-bit data value to the shift register. Since the shift register does not send any data to the master, the master would simply ignore whatever it received as a result of that transmission.

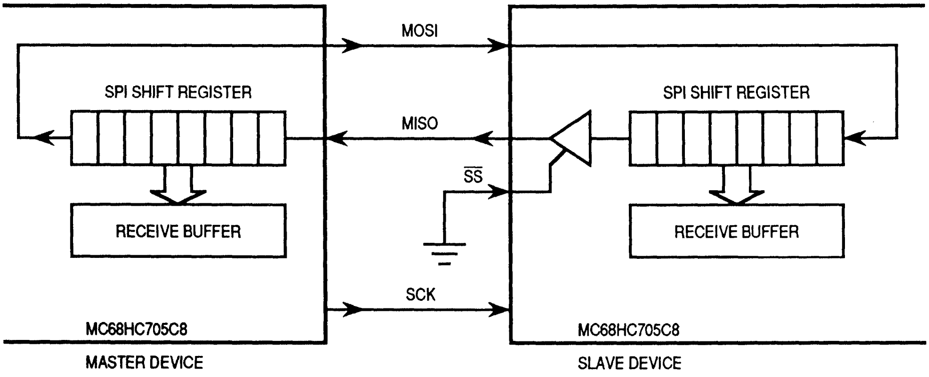


Figure 3-29. Shift Register Operation

3.7.2 Functional Description

Four I/O pins located at port D are associated with SPI data transfers. They are the serial clock (SCK–PD4), the master in/slave out (MISO–PD2) data line, the master out/slave in (MOSI–PD3) data line, and the active-low slave select ($\overline{SS}$–PD5). When the SPI system is not utilized, the four pins ($\overline{SS}$, SCK, MISO, and MOSI) are configured as general-purpose inputs (PD5, PD4, PD3, and PD2).

In a master configuration, the master start logic receives an input from the CPU (in the form of a write to the SPI data register) and originates the serial clock (SCK) based on the internal processor clock. This clock is also used internally to control the state controller as well as the 8-bit shift register. Data is parallel loaded into the 8-bit shift register (during the CPU write to SPDR) and then shifted out serially to the MOSI pin for application to the serial input line of the slave device(s). At the same time, data is applied serially from a slave device through the MISO pin to the 8-bit shift register. After the eighth shift in a transfer, data is parallel transferred to the read buffer where it is available to the internal data bus during a CPU read cycle. The SPIF status flag is used by the master and slave devices to indicate when a transfer is complete.

3.7.3 Pin Descriptions

The four I/O pins are discussed in the following paragraphs.

3.7.3.1 SERIAL DATA PINS (MISO, MOSI). The master-in slave-out (MISO) and master-out slave-in (MOSI) data pins are used for transmitting and receiving data serially: MSB first, LSB last. When the SPI is configured as a master, MISO is the master data input line and MOSI is the master data output line. In the master device, the MSTR control bit (bit 4 of the serial peripheral control register) is set to a logic one (by the program) to allow the master device to output data on its MOSI pin. When the SPI is configured as a slave, these pins reverse roles; MISO becomes the slave data output line and MOSI becomes the slave data input line.

The timing diagram of Figure 3-30 shows the relationship between data and clock (SCK). As shown in Figure 3-30, four possible timing relationships may be chosen by using control bits CPOL and CPHA. Setting CPOL is equivalent to putting an inverter in series with the clock signal. CPHA selects one of two fundamentally different clocking protocols to allow the SPI system to communicate with virtually any synchronous serial peripheral device.

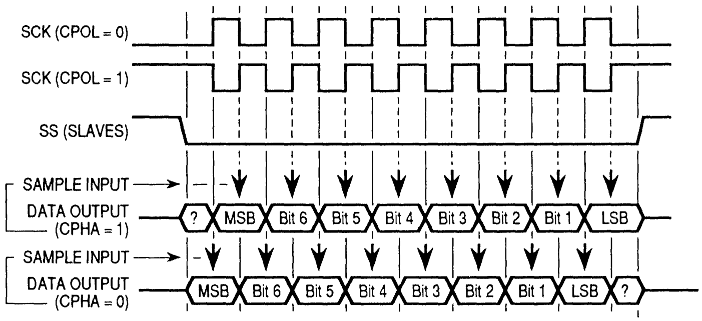


Figure 3-30. Data/Clock Timing Diagram

3.7.3.2 SERIAL CLOCK (SCK). SCK is used to synchronize the movement of data both in and out of the device through the MOSI and MISO pins. The SCK pin is an output when the SPI is configured as a master and an input when the SPI is configured as a slave.

When the SPI is configured as a master, the SCK signal is derived from the internal MCU bus clock. When the master initiates a transfer, eight clock cycles are automatically generated on the SCK pin. In both the master and slave SPI devices, data is shifted on one edge of the SCK signal and sampled on the opposite edge, where data is stable. Two bits (SPR0 and SPR1) in the SPCR (location $0A) of the master device select the clock rate. Both master and slave devices must be programmed to similar timing modes for proper data transfers, as controlled by the CPOL and CPHA bits in the SPCR.

3.7.3.3 SLAVE SELECT ($\overline{\text{SS}}$). The $\overline{\text{SS}}$ pin behaves differently on master devices than on slave devices. On a slave, this pin is used to enable the SPI slave for a transfer. On a master, the $\overline{\text{SS}}$ pin is normally pulled high externally.

3.7.4 Registers

Three registers in the SPI provide control, status, and data storage functions. These registers include the serial peripheral control register (location $0A), serial peripheral status register (location $0B), and serial peripheral data I/O register (location $0C).

3.7.4.1 SERIAL PERIPHERAL CONTROL REGISTER (SPCR). In most systems, this register (Figure 3-31) is written only once shortly after reset to initialize the SPI system.

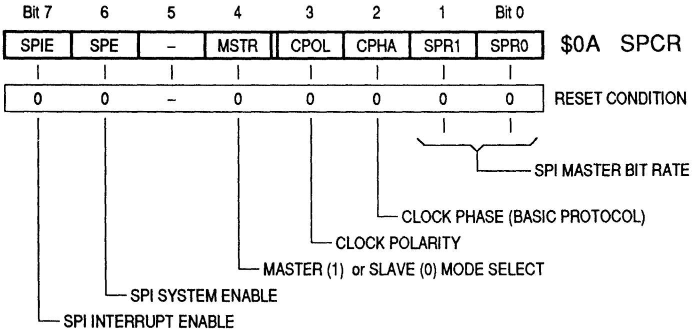


Figure 3-31. Serial Peripheral Control Register

The SPCR bits have the following functions:

SPIE

0 = SPI interrupts are disabled (the most common configuration).
1 = SPI interrupt requests are enabled if SPIF and/or MODF is set to one.

SPE

0 = SPI system is turned off.
1 = SPI system is turned on.

MSTR

0 = SPI is configured as a slave.
1 = SPI is configured as a master.

CPOL

0 = Active-high clocks selected, SCK idles low.
1 = Active-low clocks selected, SCK idles high.
(This bit is used in conjunction with the clock phase control bit to produce the desired clock-data relationship between master and slave.)

CPHA

The clock phase bit, in conjunction with the CPOL bit, controls the relationship between the data on the MISO and MOSI pins and the clock produced or received at the SCK pin. CPHA selects one of two fundamentally different clocking protocols to allow the SPI system to communicate with virtually any synchronous serial peripheral device.

SPR1/SPR0

These two serial peripheral rate bits select one of four bit rates to be used as SCK if the device is a master; they have no effect in the slave mode.

SPR1	SPR0	Internal Processor Clock Divided By	Frequency if XTAL is 4.0 MHz	Frequency if XTAL is 2 MHz
0	0	2	1.0 MHz	500.0 kHz
0	1	4	500.0 kHz	250.0 kHz
1	0	16	125.0 kHz	62.50 kHz
1	1	32	62.5 kHz	31.25 kHz

3.7.4.2 SERIAL PERIPHERAL STATUS REGISTER (SPSR).

This read-only register (Figure 3-32) contains status flags which indicate the completion of an SPI transfer and the occurrence of certain SPI system errors. The flags are automatically set by the SPI events; the flags are cleared by automatic software sequences and upon reset. In the majority of all systems, only the SPIF status bit is important.

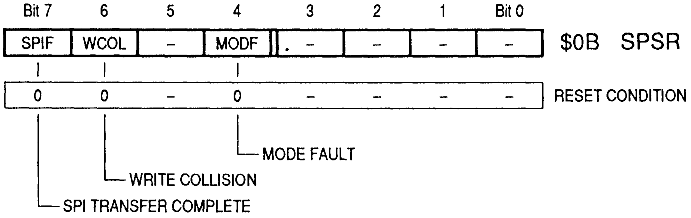


Figure 3-32. Serial Peripheral Status Register

The bits in this register have the following functions:

SPIF

When set to one, the serial peripheral data transfer flag bit notifies the user that a data transfer between the MCU and an external peripheral device has been completed. The transfer of data is initiated by the master device writing to its serial peripheral data register. SPIF is automatically cleared by reading SPSR with SPIF set, followed by an access of the SPI data register.

WCOL

The write-collision status bit notifies the user that an attempt was made to write to the serial peripheral data register while a data transfer with an external peripheral device was in progress. The transfer continues uninterrupted, and the write will be unsuccessful.

MODF

This flag is set if the $\overline{SS}$ signal goes to its active-low level while the SPI is configured as a master (MSTR = 1). In normal systems, this would never be possible. For information on how to use MODF in multimaster systems, see BR594/D, the *MC68HC705C8 Technical Summary*.

3.7.4.3 SERIAL PERIPHERAL DATA I/O REGISTER (SPDR). The SPDR (Figure 3-33) in the master MCU device is used to transmit data to and receive data from the slave device. Only a write to this register in a master will initiate transmission/reception of data. The data is then loaded directly into the 8-bit shift register where eight bits are shifted out on the MOSI pin to the slave while another eight bits are simultaneously shifted in on the MISO pin to the 8-bit shift register. At the completion of data transmission, the SPIF status bit is set. A write or read of the SPDR, after reading SPSR with SPIF set, will clear SPIF.

Bit 7	6	5	4	3	2	1	Bit 0	
								$0C SPDR

Figure 3-33. Serial Peripheral Data I/O Register

3.7.5 SPI Application Example

The example application and program are similar to the one shown in Section 2, paragraph 2.5, except the SPI function will be added.

A switch is connected to an input pin. When the switch is closed, the program will send data out to a peripheral device using the SPI function and will cause an LED connected to an output pin to light for about one second and then go out.

The peripheral device used in this application is an MC74HC595 serial-to-parallel shift register. Hardware setup, the SPI control register, and the software program will be discussed briefly.

Figure 3-34 shows the hardware connections for the SPI application example. The SPI signals at the left of the diagram come from the PGMR board (an M68HC05 PGMR, available from a Motorola distributor) or directly from the MC68HC705C8. The shift register outputs (QA–QH of the MC74HC595) will be monitored with an oscilloscope. In this example, the MISO line is not used. The shifter is selected by the general-purpose output PC3 (but could have been driven by any general-purpose output). The $\overline{SS}$ pin of the MC68HC705C8 is an input in master mode and must be tied high.

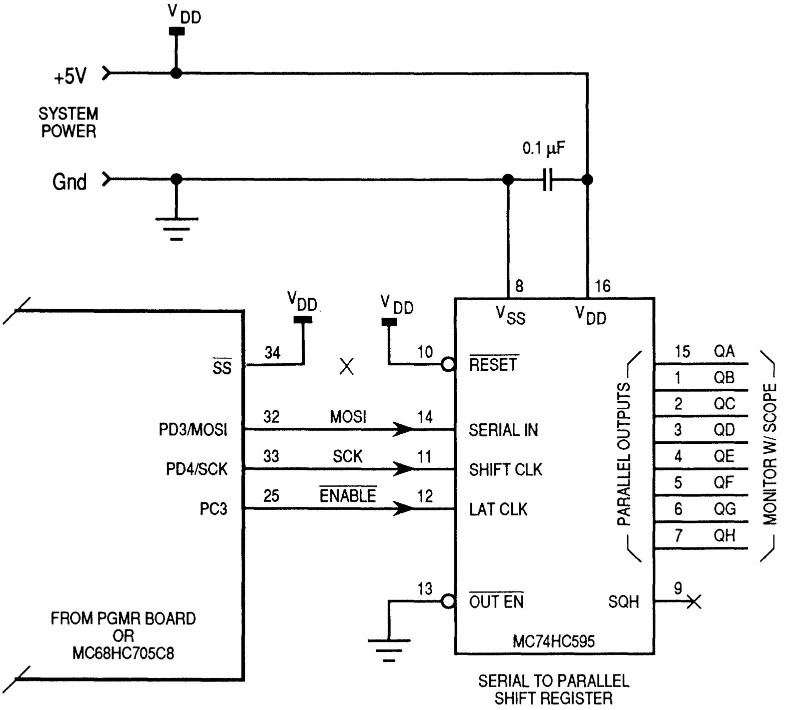


Figure 3-34. SPI Application Example Diagram

To initialize the SPI function, the SPCR (SPIE, SPE, —, MSTR, CPOL, CPHA, SPR1, SPR0) bits need to be written. For this application, the SPCR was initialized with %01010000 or $50.

SPIE = 0 No interrupts involved in this application.
SPE = 1 Enable the SPI system.
— = 0 Don't care bit.
MSTR = 1 MC68HC705C8 is the master.
CPOL = 0 Selects clock rest at low value.
CPHA = 0 MC74HC595 accepts data at rising clock edge
SPR1 = 0 Internal processor clock divide by two.
SPR0 = 0 (Shift rate = 500 kHz for a 2-MHz crystal).

The SPCR needs to be initialized once. For each transfer, there is a four-step sequence:

1) Enable the slave. In this example the PC3 general-purpose output provides the enable signal to the MC74HC595 peripheral.

2) Write data to SPDR to initiate the transfer.

3) Wait for SPIF. The slave cannot be disabled until the transfer is finished.

4) Disable the slave.

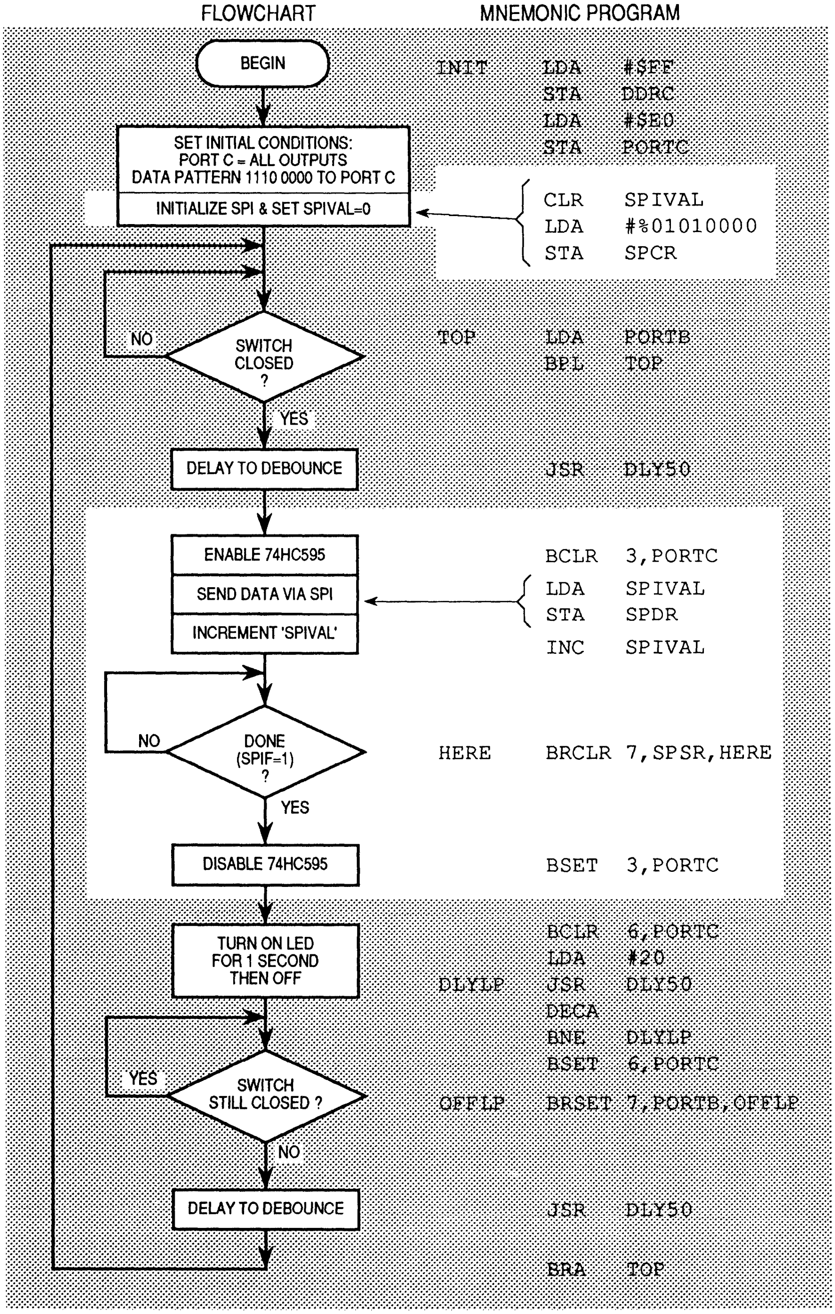
The flowchart and mnemonics for the SPI application example are shown in Figure 3-35.

Assume this application program has been assembled and downloaded to an MC68HC705C8. You can test this program by using an oscilloscope connected to the MC74HC595 parallel data outputs (pins 1–7 and 15). The program is arranged to increment the 8-bit parallel bit value each time the switch is pressed. Figure 3-36 is the complete listing for the SPI application example program.

3.8 PROGRAMMABLE TIMER

The programmable timer can be used for many purposes, including input waveform measurements, while simultaneously generating an output waveform. The architecture of the main timer is primarily a software driven system. Software can be written for measuring pulse widths and frequencies, for controlling timer output signals, or for timing delays.

The programmable timer is based on a 16-bit free-running counter preceded by a prescaler that divides the internal processor clock by four. A timer



```
INIT    LDA     #$FF
        STA     DDRC
        LDA     #$E0
        STA     PORTC
        CLR     SPIVAL
        LDA     #%01010000
        STA     SPCR
TOP     LDA     PORTB
        BPL     TOP
        JSR     DLY50
        BCLR    3,PORTC
        LDA     SPIVAL
        STA     SPDR
        INC     SPIVAL
HERE    BRCLR   7,SPSR,HERE
        BSET    3,PORTC
        BCLR    6,PORTC
        LDA     #20
DLYLP   JSR     DLY50
        DECA
        BNE     DLYLP
        BSET    6,PORTC
OFFLP   BRSET   7,PORTB,OFFLP
        JSR     DLY50
        BRA     TOP
```

NOTE: Shaded parts of this figure are identical tn Figure 2-6. Unshaded instructions were added to demonstrate the SPI system.

Figure 3-35. SPI Application Example Flowchart

```
                  ********************************************
                  * Simple 68HC05 SPI Program Example        *
                  ********************************************

0001              PORTB   EQU    $01       Direct address of port B (sw)
0002              PORTC   EQU    $02       Direct address of port C (LED)
0005              DDRB    EQU    $05       Data direction control, port B
0006              DDRC    EQU    $06       Data direction control, port C
000a              SPCR    EQU    $0A       SPIE,SPE,-,MSTR;CPOL,CPHA,SPR1,SPR0
000b              SPSR    EQU    $0B       SPIF,WCOL,-,MODF;-,-,-,-
000c              SPDR    EQU    $0C       SPI Data Register

009e              SPIVAL  EQU    $9E       One byte RAM storage location
009f              TEMP1   EQU    $9F       One byte temp storage location

0250                      ORG    $250      Program will start at $0250

0250 a6 ff        INIT    LDA    #$FF      Begin initialization
0252 b7 06                STA    DDRC      Set port C to act as outputs
                  * Port B is configured as inputs by default from reset.
0254 a6 e8                LDA    #$E8      Red & grn LED & beep off, SPI EN off
0256 b7 02                STA    PORTC     Turn off red LED
                  * Some pins of port C (my board) happen to be connected
                  * to devices which don't apply to this example program.
                  * The $E8 pattern turns off my stuff & turns off red LED

0258 3f 9e                CLR    SPIVAL    Start with 0
025a a6 50                LDA    #%01010000  SPE, MSTR, norm lo fast clk
025c b7 0a                STA    SPCR      Initialize SPI control reg

025e b6 01        TOP     LDA    PORTB     Read sw at MSB of Port B
0260 2a fc                BPL    TOP       Loop till MSB=1 (Neg trick)
0262 cd 02 86             JSR    DLY50     Delay about 50 mS to debounce

0265 17 02                BCLR   3,PORTC   Drive select of 74HC595 low
0267 b6 9e                LDA    SPIVAL    Current data to send to SPI
0269 b7 0c                STA    SPDR      Send SPI data
026b 3c 9e                INC    SPIVAL    Add one to current SPI value
026d 0f 0b fd     HERE    BRCLR  7,SPSR,HERE    Wait for SPIF to set
0270 16 02                BSET   3,PORTC   Drive select of 74HC595 hi

0272 1d 02                BCLR   6,PORTC   Turn on LED (bit-6 to zero)
0274 a6 14                LDA    #20       Decimal 20 assembles to $14
0276 cd 02 86     DLYLP   JSR    DLY50     Delay 50 mS
0279 4a                   DECA             Loop counter for 20 loops
027a 26 fa                BNE    DLYLP     20 times (20-19,19-18,.1-0)
027c 1c 02                BSET   6,PORTC   Turn LED back off
027e 0e 01 fd     OFFLP   BRSET  7,PORTB,OFFLP  Loop here till sw off
0281 cd 02 86             JSR    DLY50     Debounce release
0284 20 d8                BRA    TOP       Look for next sw closure
```

Figure 3-36. SPI Application Example Program

overflow function allows software to extend its timing capability beyond the range of 16 bits. All activities of the timer are referenced to this one free-running counter so all timer functions have a known relationship to each other. From the MCU viewpoint, physical time is represented by the count in this free-running counter and the counter can be read at any time "to tell what time it is."

The input-capture function can be used to automatically record (latch) the time when a selected transition was detected. The output-compare function can be used to generate output signals or for timing program delays.

3.8.1 Functional Description

The timer features are as follows:

- 16-Bit Free-Running Counter with Prescaler
- Overflow Flag to Extend Timing Range
- 16-Bit Output-Compare Register
- 16-Bit Input-Capture Register
- Three Interrupt Sources

The block diagram of the timer is shown in Figure 3-37.

The programmable timer capabilities are provided by using ten addressable 8-bit registers and two external pins, output level (TCMP) and edge input (TCAP). The 10 registers are as follows:

Counter High Register, location $18
Counter Low Register, location $19
Alternate Counter High Register, location $1A
Alternate Counter Low Register, location $1B
Input-Capture High Register, location $14
Input-Capture Low Register, location $15
Output-Compare High Register, location $16
Output-Compare Low Register, location $17
Timer Control Register (TCR), location $12
Timer Status Register (TSR), location $13

Because the timer has a 16-bit architecture, the counter and alternate counter, input-capture, and output-compare values are represented by two 8-bit registers. The two 8-bit registers contain the high and low byte of each timer function value (see Figure 3-38).

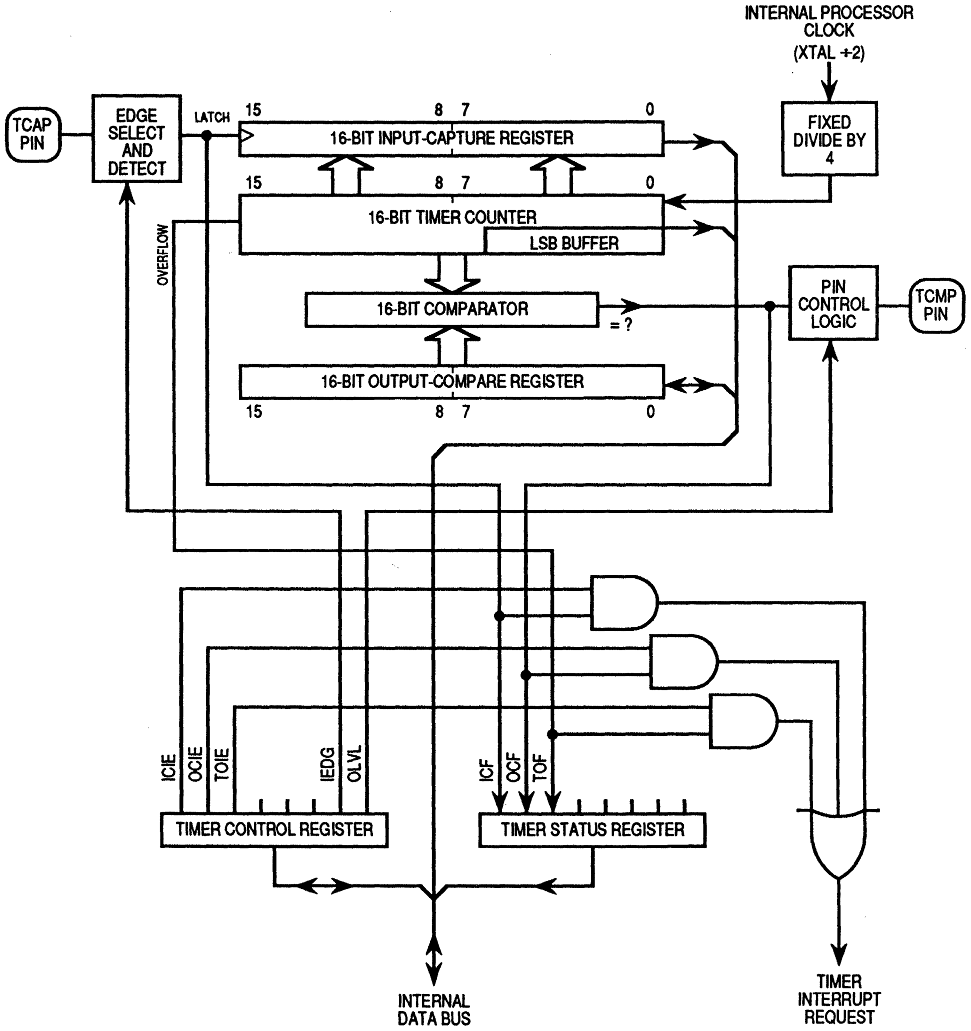
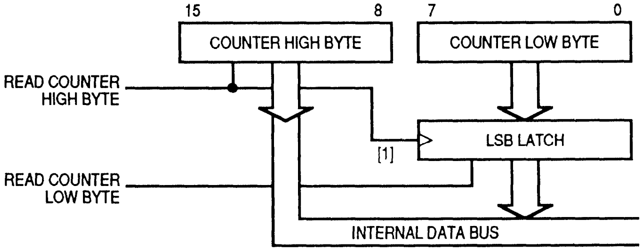


Figure 3-37. Programmable Timer Block Diagram



[1] - LSB latch is normally transparent, becomes latched when high byte of counter is read, and becomes transparent again when low byte of counter is read.

Figure 3-38. 16-Bit Counter Reads

Generally, accessing the low byte of a specific timer function allows full control of that function; however, an access of the high byte inhibits that specific timer function until the low byte is also accessed. A read from the MSB causes the LSB to be latched at the next sequential address.

NOTE

Set the I bit in the condition code register while manipulating both the high- and low-byte register of a specific timer function. This prevents interrupts from occurring between the time that the high and low bytes are accessed.

A description of each register and the external pins is given in the following paragraphs.

3.8.2 Timer Counter and Alternate Counter Registers

The 16-bit free-running counter or counter register starts from a count of \$0000 as the MCU is coming out of reset and then counts up continuously. When the maximum count is reached (\$FFFF), the counter rolls over to a count of \$0000, sets an overflow flag, and continues to count up. As long as the MCU is running in a normal operating mode, there is no way to reset, change, or interrupt the counting of this counter. This counter, which may be read at any time to "tell what time it is," is always a read-only register.

The prescaler gives the timer a resolution of 2.0 μs if the MCU crystal is 4 MHz (internal processor clock is 2.0 MHz). Including "0", the counter repeats every 65,536 counts ($FFFF = 65,535). Because the free-running counter is preceded by a fixed divide-by-four prescaler, the value in the free-running counter repeats every 262,144 internal processor clock cycles.

The double-byte free-running counter can be read from either of two locations $18–$19 or $1A–$1B. These registers are called the counter register and the counter alternate register, respectively.

NOTE

Normally, a timer read is made from the counter alternate register unless the read sequence is intended to clear the timer overflow flag.

If a read of the free-running counter register first addresses the most significant byte ($18), it causes the least significant byte ($19) to be transferred to a buffer. This buffer value remains fixed after the first most-significant-byte read, even if the user reads the most significant byte several times. This buffer is accessed when reading the free-running counter register least significant byte ($19), thus completing a read sequence of the total 16-bit counter value. The same read sequence applies to the counter alternate register. A read sequence containing only a read of the least significant byte of the free-running counter ($19) will receive the count value at the time of the read.

NOTE

In reading either the free-running counter or counter alternate register, if the most significant byte is read, the least significant byte must also be read to complete the sequence.

3.8.3 Input-Capture Concept

The input-capture function is a fundamental element of the MC68HC705C8 timer architecture. Input-capture functions are used to record the time at which some external event occurred. This is accomplished by latching the contents of the free-running counter when a selected edge is detected at the related timer input pin (edge input-TCAP pin). The time at which the event occurred is saved in the input capture register (16-bit latch). Although it may take an undetermined variable amount of time to respond to the event, software can tell exactly when the event occurred.

By recording the times for successive edges on an incoming signal, software can determine the period and/or pulse width of the signal. To measure a period, two successive edges of the same polarity are captured. To measure a pulse width, two alternate polarity edges are captured. For example, to measure the pulse width for a high-going pulse, capture the time at a rising edge and subtract this time from the time captured for the subsequent falling edge.

When the period or pulse width is known to be less than a full 16-bit counter overflow period, the measurement is very straightforward. The counter repeats every 65,536 timer clocks, which is equivalent to 262,144 internal processor clock cycles. For period or pulse widths that extend over the full 16-bit counter period, write software to keep track of the overflows of the 16-bit counter. Examples where measurement of a period or pulse width would be used are the period of a pendulum swing or the AC line frequency (to distinquish between 50 and 60 Hz).

Another important use for the input-capture function is to establish a time reference. In this case , an input-capture function is used in conjunction with an output-compare function. For example, suppose an application requires an output signal to be activated a certain number of clock cycles after detecting an input event (edge). The input-capture function would be used to record the time at which the edge occurred. A number corresponding to the desired delay would be added to this captured value and stored in the output-compare register. Since both input captures and output compares are referenced to the same 16-bit counter, the delay can be controlled to the resolution of the free-running counter, independent of software latencies. (An example of this use would be to fire a spark plug "n" microseconds after a timing pulse is sent from the engine flywheel.)

3.8.4 Input-Capture Operation

The input capture function includes a 16-bit latch, input edge detection logic, and interrupt generation logic. The latch captures the current value of the free-running counter when a selected edge is detected at the corresponding timer input pin. The edge detection logic includes a control bit (IEDG), which enables the user's software to select the polarity of edge(s) that will be recognized. The interrupt generation logic includes a status flag to indicate that an edge has been detected and a local interrupt enable bit to determine whether or not the corresponding input-capture function will generate a hardware interrupt request. See Figure 3-39.

The two 8-bit registers (locations $14–most significant byte and $15–least significant byte) comprising the 16-bit input-capture register are read-only and are used to latch the value of the free-running counter after a defined

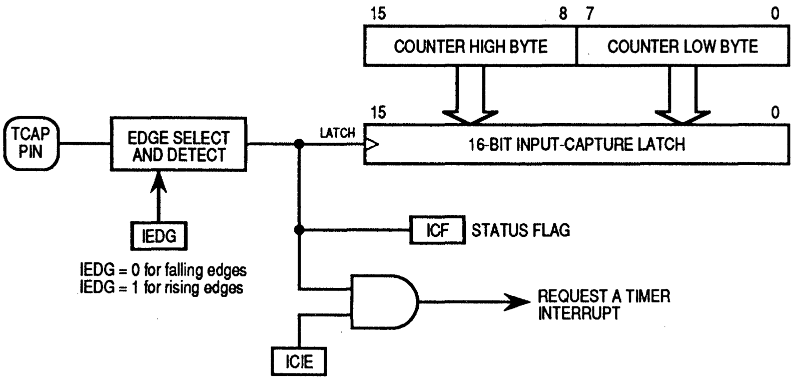

15
8
7
0
COUNTER HIGH BYTE
COUNTER LOW BYTE
15
0
TCAP PIN
EDGE SELECT AND DETECT
LATCH
16-BIT INPUT-CAPTURE LATCH
IEDG
IEDG = 0 for falling edges
IEDG = 1 for rising edges
ICF
STATUS FLAG
REQUEST A TIMER INTERRUPT
ICIE

Figure 3-39. Input-Capture Operation

transition is sensed by the corresponding input-capture edge detector. The level transition which triggers the counter transfer is defined by the input edge bit (IEDG in the timer control register).

The free-running counter contents are transferred to the input-capture register on each proper signal transition, regardless of whether the input-capture flag (ICF) is set or clear. There is an uncertainty about the exact value latched due to the resolution of the counter and synchronization delays. The input-capture register always contains the free-running counter value, which corresponds to the most recent input capture. Reset does not affect the contents of the input-capture register.

3.8.5 Output-Compare Concept

The output-compare function is also a fundamental element of the MC68HC705C8 timer architecture. Output-compare functions are used to program an action to occur at a specific time (i.e., when the 16-bit counter reaches a specific value). The value in the output-compare register is compared with the value of the free-running counter on every fourth bus cycle. When the output-compare register matches the counter value, an output is generated, which sets an output compare status flag and transfers the level of the OLVL bit to the TCMP output pin (see Figure 3-40).

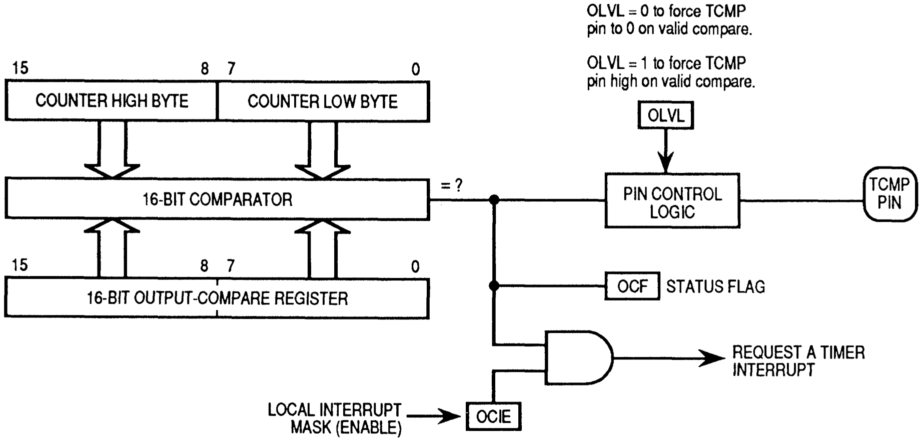

OLVL = 0 to force TCMP pin to 0 on valid compare.
OLVL = 1 to force TCMP pin high on valid compare.
15
8
7
0
COUNTER HIGH BYTE
COUNTER LOW BYTE
OLVL
16-BIT COMPARATOR
= ?
PIN CONTROL LOGIC
TCMP PIN
15
8
7
0
16-BIT OUTPUT-COMPARE REGISTER
OCF
STATUS FLAG
REQUEST A TIMER INTERRUPT
LOCAL INTERRUPT MASK (ENABLE)
OCIE

Figure 3-40. Output-Compare Operation

Change the values in the output-compare register and the output level bit after each successful comparison to control an output waveform or to establish a new elapsed timeout.

An interrupt can also accompany a successful output compare if the corresponding interrupt enable bit (OCIE) is set.

One of the easiest uses for an output-compare function is to produce a pulse of a specific duration. First, a value corresponding to the leading edge of the pulse is written to the output-compare register. The output compare is configured to automatically set the TCMP output either high or low, depending on the polarity of the pulse being produced. After this compare occurs, the output compare is reprogrammed to automatically change the output pin back to its inactive level at the next compare. A value corresponding to the width of the pulse is added to the original output-compare register value, and this result is written to the output-compare register. Since the pin-state changes occur automatically at specific values of the free-running counter, the pulse width can be controlled accurately (to the resolution of the free-running counter) independent of software latencies. By repeating the actions for generating pulses, you can generate an output signal of a specific frequency and duty cycle.

Another use of the output-compare function is to generate a specific delay. For example, suppose you want to produce a 1 millisecond delay to time programming of an EPROM byte. First, go through the initial programming steps to the point where the programming supply has been enabled (EPGM bit has been written to one). Now, read the current value of the main timer counter and add a number corresponding to 1 millisecond (XTAL = 2 MHZ, INT CLK = 1 MHz, 1 timer count = 4 μs; thus, 1 ms = 250 decimal = $FA). Write this sum to the output-compare register so that an output compare will occur when the counter gets to this value.

In this example, the actual EPROM programming time started just before the current time was read from the counter and ended after responding to the output compare and turning off EPGM. The small delays for setting up the output compare and the latency for responding to the output compare were not considered because they only make the EPROM programming time longer by a few microseconds. As you become a more advanced user of output-compare functions, you will learn how to correct these details, although it is often not necessary.

NOTE

This program would have to run from RAM since the EPROM is not usable during programming.

3.8.6 Output-Compare Operation

The output-compare register is a 16-bit register composed of two 8-bit registers at locations $16 (most significant byte) and $17 (least significant byte). The contents of the output-compare register are compared with the contents of the free-running counter once during every four internal processor clocks. If a match is found, the output-compare flag (OCF) bit is set, and the output level (OLVL) bit is clocked (by the output-compare circuit pulse) to the TCMP pin.

After a processor write cycle to the most significant byte of the output-compare register ($16), the output-compare function is inhibited until the least significant byte ($17) is also written. You must write to both bytes (locations) if the most significant byte is written first.

Because neither the output-compare flag (OCF bit) or output-compare register is affected by reset, take care when initializing the output-compare function with software. The following procedure is recommended:

1) Write to the high byte of the output-compare register to inhibit further compares until the low byte is written.

2) Read the timer status register to clear the OCF bit if it is already set.

3) Write to the low byte of the output-compare register to enable the output-compare function.

The purpose of this procedure is to prevent the OCF bit from being set between the writes to the high and low halves of the 16-bit output-compare register. A software example follows:

```
B7  16  STA  OCMPHI     Inhibit output compare
B6  13  LDA  TSR        Clear OCF bit if set
BF  17  STX  OCMPLO     Ready for next compare
```

3.8.7 Timer Control Register (TCR)

The timer control register (see Figure 3-41) is an 8-bit read/write register containing five control bits. Three of these bits control interrupts associated with the three flag bits found in the timer status register. The other two bits control 1) which edge is significant to the input-capture edge detector (i.e., negative or positive) and 2) the next value to be clocked to the TCMP output pin in response to a successful output compare.

The TCMP pin is forced low during external reset and stays low until a valid compare changes it to a high.

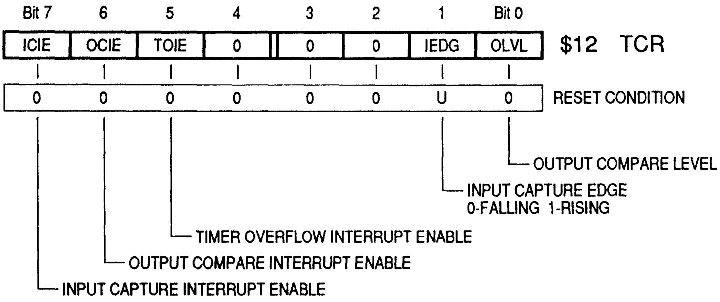


Figure 3-41. Timer Control Register

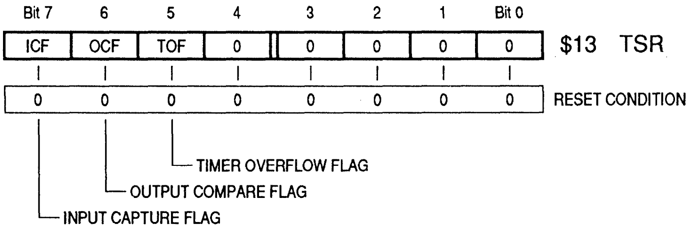
3.8.8 Timer Status Register (TSR)

The timer status register (see Figure 3-42) is an 8-bit register with three read-only bits that indicate the following status information:

1) A selected transition has occurred at the edge input (TCAP) pin with an accompanying transfer of the free-running counter contents to the input-capture register.

2) A match has been found between the free-running counter and the output-compare register.

3) A free-running counter transition from $FFFF to $0000 has been sensed (timer overflow).



Bit 7	6	5	4	3	2	1	Bit 0	
ICF	OCF	TOF	0	0	0	0	0	$13 TSR
0	0	0	0	0	0	0	0	RESET CONDITION

Figure 3-42. Timer Status Register

ICF

The input-capture flag (ICF) is set when a proper edge has been sensed by the input-capture detector. It is cleared by a processor access of the timer status register (with ICF set) followed by accessing the low byte ($15) of the input-capture register.

OCF

The output-compare flag (OCF) is set when the output-compare register contents matches the contents of the free-running counter. OCF is cleared by accessing the timer status register (with OCF set) and then accessing the low byte ($17) of the output-compare register.

TOF

The timer overflow flag (TOF) bit is set by a transition of the free-running counter from $FFFF to $0000. It is cleared by accessing the timer status register (with TOF set) and then accessing the least significant byte ($19) of the free-running counter.

NOTE

The counter alternate register contains the same value as the free-running counter but reading the alternate register does not clear TOF; therefore, this alternate register should be used to read the timer counter in all cases except when intending to clear TOF. This will avoid the possibility of the TOF being unintentionally cleared.

3.8.9 Timer Application Example

Figure 3-43 shows an example program to produce a 10-second delay after the timer counter is read. In this case, the timer counter and the output-compare functions are used in the software program.

The two key programming instructions that you should note are 1) the read and/or write instructions at the alternate counter and output-compare registers and 2) the addition of 16-bit numbers.

3.9 STOP/WAIT INSTRUCTION EFFECTS

The STOP and WAIT instructions put the MC68HC705C8 MCU into low power-consumption modes. These instructions also affect the programmable timer, the SCI, and the SPI systems. A STOP/WAIT flowchart is shown in Figure 3-44.

3.9.1 Low Power-Consumption Modes

The STOP instruction places the MC68HC705C8 in its lowest power-consumption mode. In the STOP mode, the internal oscillator is turned off, causing all internal processing to be halted. During the STOP mode, the I bit in the condition code register is cleared to enable external interrupts. All other registers and memory remain unaltered, and all I/O lines remain unchanged. This state continues until an external interrupt ($\overline{\text{IRQ}}$) or $\overline{\text{RESET}}$ is sensed, at which time the internal oscillator is turned on. The external interrupt or reset causes the program counter to vector to memory location $1FFA and $1FFB or $1FFE and $1FFF. These locations contain the starting address of the interrupt or reset service routine, respectively.

The WAIT instruction also places the MC68HC705C8 in a low power-consumption mode, but the WAIT mode consumes somewhat more power than the STOP mode. In the WAIT mode, all CPU processing is stopped;

```
                    ******************************************
                    * Simple 68HC05 Timer Program Example      *
                    ******************************************
0006                DDRC    EQU   $06        Data direction control, port C
0002                PORTC   EQU   $02        Direct address of port C (LED)
0016                OCMPHI  EQU   $16        Output compare high reg.
0017                OCMPLO  EQU   $17        Output compare low reg.
0013                TSR     EQU   $13        ICF,OCF,TOF,0;0,0,0,0
00a0                TENSEC  EQU   $A0        Used to count 39 out compares
00a1                TEMP    EQU   $A1        One byte temp for 16 bit OCMP add

0350                        ORG   $350
0350 a6 40          INIT    LDA   #%01000000  Make DDR bit for LED a one
0352 b7 06                  STA   DDRC       So Red LED pin is an output
0354 a6 40          BEGIN   LDA   #%01000000  Port C bit 6 is red LED
0356 b8 02                  EOR   PORTC      Toggle red LED on PGMR board
0358 b7 02                  STA   PORTC      Red LED will change every 10 Sec
035a a6 27                  LDA   #39        10 sec = 38 rev + 9,632 ticks
035c b7 a0                  STA   TENSEC     Counter for timer out compares

 *********************************************************************
 * For XTAL=2MHz (Int proc. clk=1MHz) Timer ÷4 makes 1 count = 4µS    *
 * Counter rolls from $FFFF to 0 every 65,536 counts (262.144 mS)     *
 * 10 Sec ÷ 262.144 mS = 38 revs of timer & 9,632 counts remainder    *
 * 10 Sec = 2,500,000 counts @ 4µS/count.  38 * 65,536 = 2,490,368    *
 * 2,500,000 - 2,490,368 = 9632.  9632 (decimal) = $25A0              *
 *                                                                    *
 * To time 10 Sec, read initial count, add 9632 (remainder count)     *
 * store to out compare reg ("schedule a compare").  When OCF flag =1 *
 * clear it & next compare will occur when timer counts 65,536 counts *
 * count the first compare plus 38 more compares (full revs).         *
 *********************************************************************

035e a6 a0                  LDA   #$A0       Lower half hex equiv of 9632
0360 bb 17                  ADD   OCMPLO     Low half of a 16 bit add
0362 b7 a1                  STA   TEMP       Temp. store until OCMPHI is added
0364 a6 25                  LDA   #$25       Upper half hex equiv of 9632
0366 b9 16                  ADC   OCMPHI     High half of 16 bit add (w/ carry)
0368 b7 16                  STA   OCMPHI     Update OCMP hi
036a b6 a1                  LDA   TEMP       Get previous saved OCMP low
036c b7 17                  STA   OCMPLO     Update OCMP lo after OCMP hi

 *********************************************************************
 * You add low half first due to possible carry, then add high byte   *
 * including any carry (ADC).  You should update out compare high     *
 * byte first to avoid an erroneous compare value (compare lockout    *
 * after OCMPHI till OCMPLO prevents this potential problem.          *
 *********************************************************************

036e 0c 13 fd  LOOP     BRCLR 6,TSR,LOOP  Checks for out comp. flag
0371 b6 17              LDA   OCMPLO      To clear OCF flag
0373 3a a0              DEC   TENSEC      Ten seconds count down
0375 26 f7              BNE   LOOP        Loop until 10 sec done
0375 20 db              BRA   BEGIN     Repeat so PC6 toggles /10 Sec
```

Figure 3-43. Timer Application Example Program

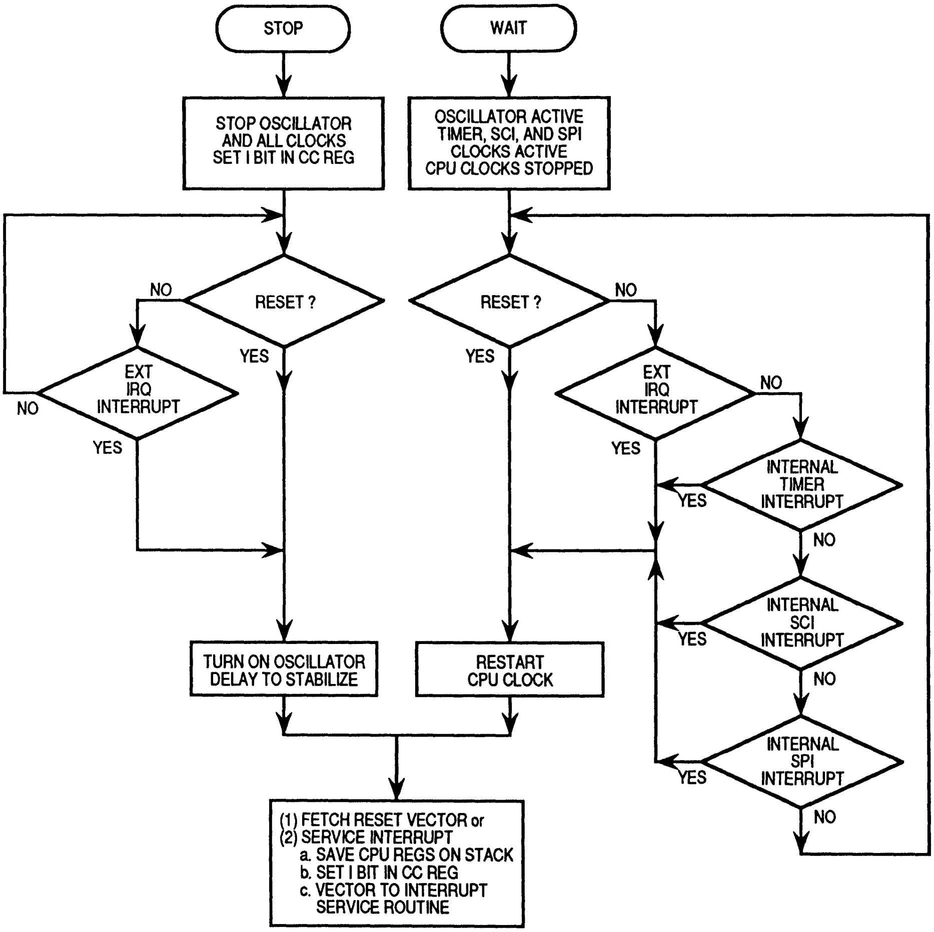

STOP
WAIT
STOP OSCILLATOR AND ALL CLOCKS SET I BIT IN CC REG
OSCILLATOR ACTIVE TIMER, SCI, AND SPI CLOCKS ACTIVE CPU CLOCKS STOPPED
RESET ?
NO
YES
EXT IRQ INTERRUPT
INTERNAL TIMER INTERRUPT
INTERNAL SCI INTERRUPT
INTERNAL SPI INTERRUPT
TURN ON OSCILLATOR DELAY TO STABILIZE
RESTART CPU CLOCK
(1) FETCH RESET VECTOR or
(2) SERVICE INTERRUPT
a. SAVE CPU REGS ON STACK
b. SET I BIT IN CC REG
c. VECTOR TO INTERRUPT SERVICE ROUTINE

Figure 3-44. STOP/WAIT Flowchart

however, the internal clock, the programmable timer, SPI and SCI systems (if enabled) remain active. During the WAIT mode, the I bit in the condition code register is cleared to enable all interrupts. All other registers and memory remain unaltered, and all parallel I/O lines remain unchanged. This state continues until any interrupt or reset is sensed. At this time, the program counter is loaded with the interrupt vector at memory location $1FF4–$1FFF, which contains the starting address of the interrupt or reset service routine.

3.9.2 Effects on On-Chip Peripherals

The STOP instruction causes the oscillator to be turned off, which halts all internal CPU processing as well as the operation of the programmable timer, SCI, and SPI. The oscillator starts again when an external interrupt ($\overline{\text{IRQ}}$) or $\overline{\text{RESET}}$ occurs.

3.9.2.1 TIMER ACTION DURING STOP MODE. When the MCU enters the STOP mode, the timer counter stops counting (the internal processor clock is stopped). It remains at that particular count value until an interrupt or reset occurs. If the interrupt is an external low on the $\overline{\text{IRQ}}$ pin, the counter resumes from its stopped value as if nothing had happened. If a reset occurs, the counter is forced to $FFFC.

3.9.2.2 SCI ACTION DURING STOP MODE. When the MCU enters the STOP mode, the baud rate generator driving the receiver and transmitter is stopped, which halts all SCI activity.

If the STOP instruction is executed during a transmitter transfer, that transfer is halted. When the STOP mode is exited, that particular transmission resumes if the exit is the result of a low input to the $\overline{\text{IRQ}}$ pin. Since the STOP mode interferes with SCI character transmission, make sure that the SCI transmitter is idle when the STOP instruction is executed.

If the receiver is receiving data when the STOP instruction is executed, received data sampling is stopped (baud rate generator stops), and the remainder of the data is lost. The STOP mode should not be used while SCI characters are being received.

3.9.2.3 SPI ACTION DURING STOP MODE. When the MCU enters the STOP mode, the bit rate generator driving the SPI stops, halting all master mode SPI operation. Thus, the master SPI is unable to transmit or receive data. If the STOP instruction is executed during an SPI transfer, that transfer is halted until the MCU exits the STOP mode (if the exit resulted from a logic low on the $\overline{\text{IRQ}}$ pin). If the STOP mode is exited by a reset, then the appropriate control/status bits are cleared, and the SPI is disabled.

If the device is in the slave mode when the STOP instruction is executed, the slave SPI will still operate. It can still accept data and clock information in addition to transmitting its own data back to a master device. At the end of a transmission with a slave SPI in the STOP mode, no flags are set until a logic low $\overline{\text{IRQ}}$ input results in an MCU "wake up."

When the MCU enters the STOP mode, all enabled output drivers (TDO, TCMP, MISO, MOSI, and SCK ports) remain active. Any sourcing currents from these outputs will be part of the total supply current required by the device.

3.9.2.4 WAIT MODE EFFECTS. When the MCU enters the wait mode, the CPU clock is halted. All CPU action is suspended; however, the timer, SCI, and SPI systems remain active. An interrupt from the timer, SCI, or SPI (in addition to a logic low on the $\overline{\text{IRQ}}$ or $\overline{\text{RESET}}$ pins) will cause the processor to resume normal processing.

The wait mode power consumption depends on how many systems are active. The power consumption will be greatest when all the systems (timer, TCMP, SCI, and SPI) are active. The power consumption will be least when the SCI and SPI systems are disabled (timer operation cannot be disabled in the wait mode). If a nonreset exit from the wait mode is performed (e.g., timer overflow interrupt exit), the state of the remaining systems will be unchanged. If a reset exit from the wait mode is performed, all systems revert to the (disabled) reset state.

3.10 OTPROM/EPROM PROGRAMMING

The OTPROM or EPROM programming technique is used to load a user program into the MC68HC705C8 MCU OTPROM or EPROM. This type of programming is accomplished via a bootstrap mode of operation.

3.10.1 Erasing

MC68HC705C8 EPROM MCUs are erased by exposure to a high-intensity ultraviolet (UV) light with a wavelength of 2537 angstrom. The recommended dose (UV intensity × exposure time) is 15 Ws/cm2. UV lamps should be used without shortwave filters, and the EPROM MCU should be postioned about one inch from the UV lamps.

MC68HC705C8 one-time programmable ROM (OTPROM) MCUs are shipped in an erased state and are packaged in an opaque plastic package; thus, erasing operations cannot be performed on OTPROM MCUs.

3.10.2 Programming

Programming operations are controlled by software-accessible control bits. The software program which programs the internal EPROM/OTPROM is located in either the on-chip bootstrap ROM or internal RAM.

The first programming method uses a program in the bootstrap ROM to read information from an external 8K by 8 EPROM and to program this information into the on-chip EPROM/OTPROM. The external EPROM is connected to I/O port pins of the MC68HC705C8. In this programming method, information to be programmed into the internal EPROM/OTPROM is first programmed into the external EPROM using an industry-standard PROM programmer.

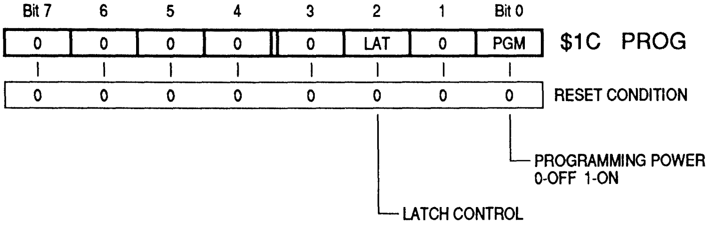
A second programming method allows user programs developed on a personal computer to be downloaded to the MC68HC705C8 for programming into the on-chip EPROM/OTPROM. This method eliminates the extra steps needed to program a separate 8K by 8 EPROM. A small program that runs on the personal computer is available through the Motorola FREEWARE bulletin board service (BBS) and can be downloaded for the price of the phone call. This method is explained in Section 4 of this applications guide.

3

Both methods described for programming the on-chip EPROM/OTPROM ultimately use a software program running in the MCU that is being programmed. The programming software uses the program register (PROG) to control the EPROM programming process.

3.10.3 Program Register

The program register (see Figure 3-45) is used for PROM programming.



Bit 7	6	5	4	3	2	1	Bit 0	
0	0	0	0	0	LAT	0	PGM	$1C PROG
0	0	0	0	0	0	0	0	RESET CONDITION

Figure 3-45. Program Register

LAT

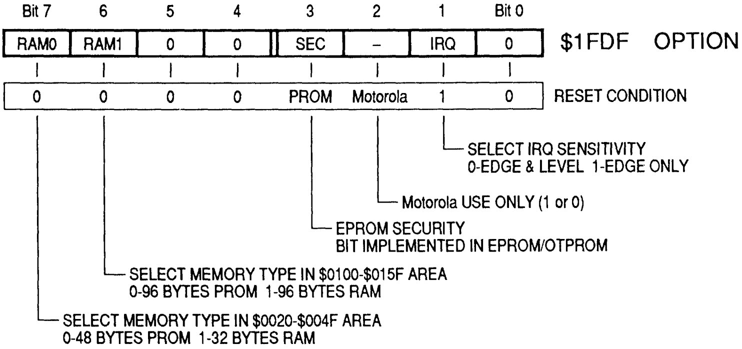
Prior to a PROM write operation, set the latch (LAT) bit. This enables the PROM data and address buses to be latched for programming a PROM location. Reset clears the LAT bit. When the LAT bit is cleared, PROM data and address buses are unlatched for normal CPU operations. This bit, which is both readable and writable, must be cleared to allow PROM read operations.

PGM

When the program (PGM) bit is set, V_{PP} power is applied to the PROM for programming mode of operation. Reset clears the PGM bit. This bit, which is readable, is only writable when the LAT bit is set. If the LAT bit is cleared, the PGM bit cannot be set.

3.10.4 Option Register

The option register (see Figure 3-46) is used to select memory RAM/ROM configurations, enable PROM security, and select the MCU $\overline{\text{IRQ}}$ pin sensitivity.



Bit 7	6	5	4	3	2	1	Bit 0	
RAM0	RAM1	0	0	SEC	–	IRQ	0	$1FDF OPTION
0	0	0	0	PROM	Motorola	1	0	RESET CONDITION

SELECT IRQ SENSITIVITY
0-EDGE & LEVEL 1-EDGE ONLY
Motorola USE ONLY (1 or 0)
EPROM SECURITY
BIT IMPLEMENTED IN EPROM/OTPROM
SELECT MEMORY TYPE IN $0100-$015F AREA
0-96 BYTES PROM 1-96 BYTES RAM
SELECT MEMORY TYPE IN $0020-$004F AREA
0-48 BYTES PROM 1-32 BYTES RAM

Figure 3-46. Option Register

RAM0

The RAM0 bit determines the amount and type of memory in the $0020–$005F area.

0 = 48 bytes of PROM ($0020–$005F)

1 = 32 bytes of RAM ($0030–$005F)

When RAM is selected by RAM0 = 1, the 16 bytes from $0020–$002F are unused. This bit is readable and writable at all times, allowing selection of the desired memory configuration during program execution. Reset clears the RAM0 bit.

RAM1

The RAM1 bit determines the type of memory in the \$0100–\$015F area.

0 = 96 bytes of PROM
1 = 96 bytes of RAM

This bit is readable and writable at all times, allowing selection of the desired memory configuration during program execution. Reset clears the RAM1 bit.

SEC

The SEC bit is implemented as a PROM bit. During PROM programming, the SEC bit is set to enable the security feature (to disable the bootloader). This bit is normally cleared (security disabled) for an OTPROM device. For an EPROM device, clearing is accomplished by exposing the EPROM to UV light until the SEC bit is erased.

Bit 2

Factory use (logic one or logic zero).

IRQ

When the IRQ bit is set (logic one), the $\overline{\text{IRQ}}$ pin is negative edge and level sensitive. When the IRQ bit is cleared (logic zero), the $\overline{\text{IRQ}}$ pin is negative edge sensitive. Reset sets the IRQ bit. The IRQ bit can only be written once following each reset.

SECTION 4
APPLICATIONS

This section discusses the development of an application (home thermostat project) based on a microcontroller. A typical MCU application involves hardware development, software development, and mechanical development. Though separate to some degree, all elements must work together as a system; thus, everyone working on the project should be somewhat familiar with the requirements of each element.

The principles of systematic project management, including planning, review, prototyping, and testing, still apply. Although genius and unusual creativity are assets to a microcontroller designer, they are not a requirement. The majority of MCU applications result from simple systematic development. Due to the nature of MCUs, applications based on an MCU often include noteworthy features that cannot be found on similar products which do not use an MCU.

In this applications guide, we assume some knowledge of the traditional mechanical and electrical aspects of a project. What is new is the software program that allows the MCU to perform the desired functions of the application. On-chip peripherals that can be configured and controlled by program instructions are also a new concept.

When residential electricity became common, house plans required additional pages to document the location of switches and outlets. The idea of how electricity went from one place to another was foreign to the architects of the day. A new system of symbols and conventions had to be developed.

MCU-based application projects are essentially the same as mechanical or discrete logic projects except for the addition of software programming. Software programming is not entirely an added design task because the programmable nature of an MCU simplifies the hardware aspects of the project.

The normal order of events in MCU-based projects is as follows:

1) Proposal — A marketing and/or design group proposes preliminary requirements of a project to satisfy customer demand.

2) Specification — This step defines limits of operation but should not identify internal components, preventing selection of the most cost-effective solution to a problem.

3) Breadboarding — This procedure is primarily a hardware activity although some software is normally required to verify the accuracy of the hardware design.

4) Software Development — This step involves planning and implementation of software programs. The programmer must know how the system is electrically interfaced to components outside the MCU because software programs control the operation of these external components.

5) System Integration — This procedure involves putting together finished (preliminary) software and hardware.

6) Testing — This step is a design verification process.

In practice, the steps occur in parallel to some degree, and some changes normally occur during the development which impact all of the steps. In this applications guide, we assume you are familiar with traditional design methods; therefore, we will only discuss how MCU-based methods differ from traditional methods.

The first area of difference is in the hardware design where the flexibility of the software-driven MCU simplifies the connection of external circuitry. Signal polarity and timing are easily controlled by software to match the needs of external components. The hardware design consists of connecting peripheral devices to general-purpose I/O lines and of checking the ability of software to control the connected devices.

The second and most significant area of difference between MCU-based projects and discrete logic projects is the area of software development. The preparation of programs replaces the development of complex logic circuits. Instead of laboring over complex wire-wrapped breadboards with an oscilloscope, the programmer sits at a computer terminal and develops sets of computer instructions.

4.1 HARDWARE DEVELOPMENT METHODS

When a project has been selected, determine what hardware will be required for the final design (input and output devices and power supply) and what hardware can be used to make the prototype (substitutions such as potentiometers for temperature sensors).

Two approaches can be used to develop a hardware circuit (breadboarding) for a system based on an M68HC05 MCU. You can use an M68HC05 PGMR board, or you can wire a complete circuit on another board with a socket for the MCU. The PGMR board approach is the fastest since the basic wiring to the MCU is already done. The complete circuit with a socket for the MCU has the advantage of not having to worry about interference between PGMR board functions and application requirements.

Since the PGMR board is also used to program information into the EPROM in the MCU, there are a few areas where some conflict may occur between the planned application and components on the PGMR board. The areas are small and usually easy to avoid. For example, the port D pins of the MCU are connected to switches on the PGMR board. To use these pins, you would turn off the switches so that there is no conflict with the components of your application.

Also the PGMR board can be used with other members of the M68HC05 Family to increase your development choices. In addition to the MC68HC705C8 8K EPROM device, the PGMR can also operate with the MC68HC805C4 4K EEPROM device or the MC68HC05A6 4K ROM + 2K EEPROM device. Each of these devices supports a slightly different approach to development.

With the EPROM approach (MC68HC705C8), you would write a software program, transfer this program into the EPROM in the MCU, and reset the MCU to execute the program. When you discover a mistake or want to make a change, you remove the MCU from the PGMR board and erase the EPROM with an ultraviolet (UV) light source. After the MCU is erased, you can program the modified program into it and continue debugging (finding errors).

After a program is developed with a windowed EPROM, you can program the working software program into any of several OTP MCUs for use in your finished products. The OTP MCU is identical to the windowed device used for development, except that it is packaged in a less expensive plastic package. Since this plastic package is opaque, you cannot erase the on-chip EPROM after it has been programmed.

The MC68HC805C4 has 4K bytes of electrically erasable PROM (EEPROM), which allows easier erasure of programs during development (EEPROM does not have to be erased with UV light). In most other respects this MCU is the same as the MC68HC705C8 OTPROM MCU. Thus, programs can be developed with the MC68HC805C4 and later programmed into less expensive MC68HC705C8 OTP MCUs for production quantities.

The MC68HC05A6 is also similar to the MC68HC705C8 except that it has 2K bytes of EEPROM and a 4K-byte ROM monitor program instead of an 8K-byte EPROM. This on-chip monitor program allows you to interactively develop and check programs; thus, you can develop a program on a personal computer, transfer it into the EEPROM in the MC68HC05A6, and try the program using the monitor commands of the on-chip monitor program. When a mistake is found, you can often make the change with the single-line assembler command of the monitor and retry the program immediately. This was the method used to develop the thermostat application example. Although we could have developed this application using the MC68HC705C8, it would have taken slightly longer.

Motorola produces a line of low-cost (about $500) evaluation boards (EVMs) which can be used for high-speed interactive development. To use this development approach, you would build a prototype of your system with a socket where the MCU will go. Instead of an MCU, you would connect the EVM into this socket. The EVM emulates the actions of a real MCU but allows visibility into the internal activities of the MCU.

Some of the possible uses for an EVM include examination and modification of memory locations, executing a user program until a certain instruction is found, or looking at a program in mnemonic form. You can also trace individual instructions and look at the contents of registers and memory before and after executing each instruction.

4.2 SOFTWARE DEVELOPMENT METHODS

The development of programs for MCU-based systems requires the use of slightly different techniques from those used with hardware-based systems. MCU-based systems are programmed with instructions which control the MCU; whereas, hardware-based systems are programmed by changing wired connections. This section describes program development techniques for MCU-based systems.

A program is a series of instructions for the MCU. The program gives the MCU alternatives to transact, depending on what it learns as the result of executing previous instructions.

For instance, to determine if a thermostat should operate the compressor or the heater, we might program it as follows:

1) Read the existing temperature.
2) Read the desired temperature setting.
3) Compare these two readings.
4) If existing is less than desired, operate heater.
5) If existing is more than desired, operate compressor.

To write a program, you can draw a flowchart to show the decision process that must be performed to accomplish a specific task. Flowcharts are not always necessary; sometimes a list of steps will do, depending upon the application complexity.

In general, programming requires planning and developing rules, algorithms, or flowcharts. Programs evolve by repeating the following steps several times:

1) Generate the source file (the program in mnemonic form).
A development station (usually a personal computer) is used to generate a text file. This text file, the source of the data to be run by the MCU, is called the "source program." This text file is for the convenience of the programmer since the MCU understands only 8-bit bytes of encoded information. This text representation makes it easier to develop the program. Previously, programs for computers had to be in binary form, the native code of the computer.

2) Translate the source file.
The text file is then translated into a binary object file (or S-record encoded object file) by an assembler. This assembler program runs on the development station, not on the MCU. The assembler does not usually directly generate the final binary file (i.e., the object code or executable file for the MCU) since this file has to be transferred from the development station to the MCU. The transfer process can create errors from external electrical noise. Motorola has a file transfer form which encodes the MCU object file into ASCII data with a checksum for error detection. This encoding is referred to as Motorola "S-records" or "S1-S9" records.

3) Transfer the object file into the MCU.
The final step in developing MCU-based systems is to transfer the S-record or binary file (the MCU program) to the MCU itself. We can

take the binary or S-record file and send it to program an external EPROM in an EPROM programmer; send it to an EPROM programmer to program the MCU directly (not all EPROM programmers support this); or send the file to the MCU in bootstrap mode and have the MCU program itself. In all cases, the S-record file is used and is translated to binary during the programming process so the MCU can use the object file.

4.2.1 Freeware

Motorola has an electronic bulletin board system (BBS) dedicated to support Motorola microprocessor units (MPUs) and microcontroller units (MCUs). "Freeware," the name for this BBS, is on-line 24 hours a day, except when system maintenance is required. The following is a sample of the available freeware topics:

8-Bit MCUs
16- and 32-Bit MPUs
Evaluation Boards (EVBs) and Evaluation Modules (EVMs)
Development Systems (HDS-200 and HDS-300)
IBM-PC Software Tools (assemblers, etc.)
Conference and Special Interest Groups

To use the BBS, you need to obtain the following hardware and software items:

1) A 1200/2400 baud modem
2) A terminal or personal computer (PC) with communications software (e.g. Kermit, ProComm, etc.)
3) A telephone line

Use the following procedure to log onto the freeware line:

1) Set systems character format to 8-bit, no parity, 1 stop bit.
2) Dial (512) 891-3733 or (512) 891-FREE.
3) A series of questions will appear. Enter the requested information to log on. You are now a registered user.
4) Follow the menus for the desired functions (e.g., download, upload, mail, conferences, etc). On-line help is also available.

4.2.2 Third-Party Software

Many third-party vendors sell assemblers to translate mnemonic text files into machine-readable files. These assemblers are similar to the free assembler available on the Freeware BBS except that the third-party assemblers offer additional features.

One common feature is the ability to use macros. Macros are sets of instructions used repeatedly in a program. A set of instructions can be typed into the program, declared as a macro, and be given a name. When this set of instructions is needed again, you would type the name of the macro where an instruction mnemonic would normally go. The assembler recognizes the macro name and inserts the previously defined set of instructions at that point into the machine-readable object file. Macros improve programmer productivity and often improve the readability of the assembly-language listing.

A simulator is a software program that runs on a personal computer (or other computer system). The simulator emulates the behavior of an MCU in the same way you would play computer (see **2.6.2 Playing Computer**). Although a simulator does not operate as fast as the actual MCU, it does operate much faster than you could play computer.

In a typical simulator, the computer screen will display windows showing current and recent contents of memory and registers as well as the condition of I/O pins and peripheral systems. These displays help a programmer understand the operation of a program under development better than the other methods of software development.

A simulator can show internal conditions that are not visible from outside the MCU. In other development methods, the programmer has to deduce this information indirectly. Two disadvantages of the simulator approach are operating speed and accuracy of emulation. In terms of speed, the simulator runs much slower than a real MCU would (although this is often fast enough so the programmer does not notice any problems). Since simulators are based on a software emulation of specified MCU operation, there can be subtle differences between the way the simulator behaves and the way a real MCU behaves. Ideally, these differences are small enough not to be significant; in reality, the differences sometimes cause problems.

A compiler is similar to an assembler, but it translates a higher level language into a machine-readable object file (rather than translating mnemonic assembly language). One common high-level language is "C."

The object of programming in C or some other high-level language instead of assembly language is to improve productivity and to avoid learning the assembly language of several different MCUs. The compiler translates the high-level language instructions into a machine-readable object file for a particular MCU.

The greatest disadvantage of using a high-level language and a compiler is the significant inefficiency introduced in translating to the MCU machine language. The degree of inefficiency depends on the power of the MCU instruction set and the task being performed. The M68HC05 has a relatively small instruction set compared to a mainframe or personal computer; thus, it is difficult and inefficient to use C language instructions in this MCU.

The inefficiency of using C language instructions also affects timing of I/O operations. For some applications where very fine control of timing is important, it is better to use assembly language than to use C. Inefficient programs also require more memory to perform a task.

For many applications, the speed of the CPU is so great compared to the requirements of the application that the inefficiencies of high-level language are unimportant. Present-day MCUs often have enough on-chip memory so that program size may be unimportant. Using high-level language with the M68HC05 is not recommended in most cases. However, at least one good C compiler is available for the M68HC05. If you want to use high-level languages for Motorola MCUs, you can get a list of names and addresses of third-party vendors and products from a local Motorola representative or by calling the freeware BBS.

4.3 THERMOSTAT PROJECT DETAILS

The major steps for the project to be developed are as follows:

1) Select the application — in this case, a home thermostat.

2) Define the functions desired for the thermostat.
Read/display existing indoor/outdoor temperature
Enter/display desired indoor/outdoor temperature
Enter/display time of day
Select heating or cooling
Operate heater or compressor

3) Determine the hardware required based on the functions.
   A microcontroller (MC68HC705C8)
   Temperature sensing devices
   A/D converters (MC145041)
   Keypad
   Display
   Relays/relay drivers
   Audible alarm device
   Pullup resistors
   Bypass capacitors
   Power supply
   Circuit board

4) Develop simple programs to test the hardware circuits. Develop the main program for the desired functions. The program(s) to be written for this project are as follows:
   A program to test the audible alarm
   A program to test the display
   A program to test the display and keypad
   A program to test the basic software organization

The programs written for this thermostat application will be written in assembly language on a PC using the MCU instruction set commands. An assembler program contained in the PC memory will translate the programs into machine language — i.e., a series of binary codes of "0" and "1" which the MCU understands. This code will be put into the OTPROM or EPROM to be debugged.

4.3.1 Hardware Details

The best way to learn about MCUs is to try this application example thermostat project and develop additional projects in your area of interest. Even if you choose not to duplicate this thermostat project, you can still benefit from studying the documentation in this example.

Figure 4-1 is the schematic diagram for the thermostat project. For development, the MC68HC705C8 is being replaced by the M68HC05 PGMR board. In this schematic diagram, only the I/O circuitry is shown. To see the other MCU connections, refer to the schematic diagram of the PGMR board in the *Programmer Board User's Manual* included with the PGMR board.

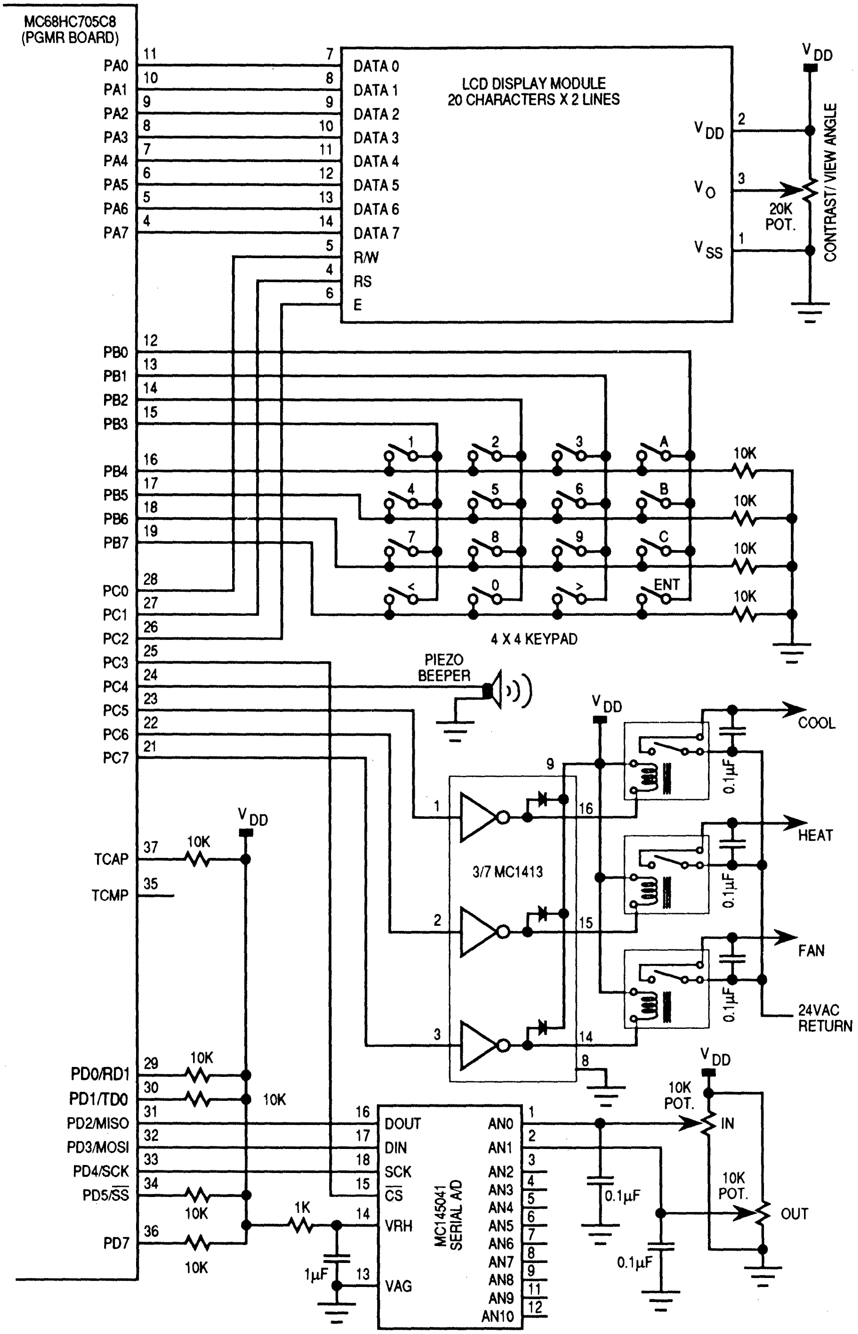


Figure 4-1. Thermostat Project Schematic Diagram

During development, it was convenient to use potentiometers rather than temperature sensors because doing so allowed us to simulate temperature changes. In the final application, we would use an actual temperature sensor such as that shown in Figure 4-2.

The LCD display is used to show the keypad entries of time-of-day, the temperature limits, the current temperature, and the selection of heating or cooling operation. The keypad can be a 4×4 array or larger. An audible alarm can be used along with the display, if desired.

The project parts list is shown in Table 4-1. Only the parts not commonly available are listed.

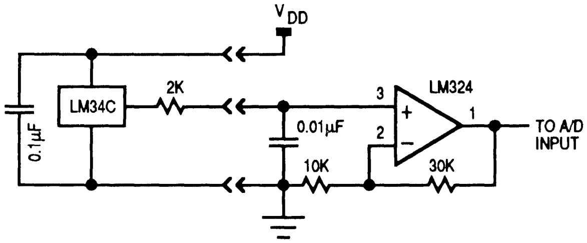


Figure 4-2. Precision Temperature Sensing Circuit

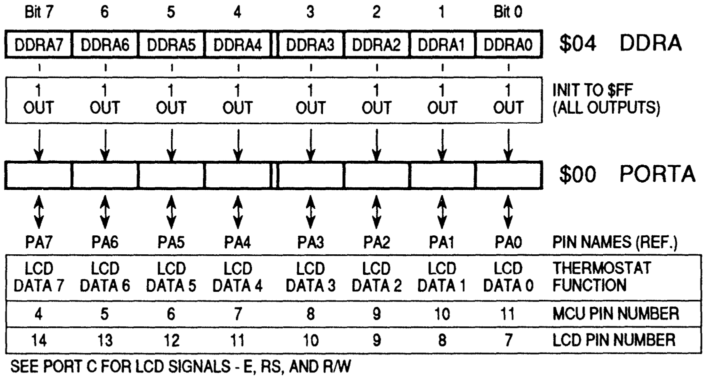
Table 4-1. Thermostat Project Parts List

Item and Description	Suggested Source
LCD Display Module — 20 Characters by 2 Lines	Digi-Key Wholesale OP220-ND
Keypad — 4 by 4 Matrix of Momentary Push-Button Switches	Any
Piezo Beeper — Solid State Buzzer	Radio Shack 273-060A
A/D Converter — Serial Interface to SPI	Motorola — Special Functions MC145041
Relay Driver — Translates 0–5 V MCU Signals to High Current Inductive Load Drive	Motorola — Interface MC1413 or ULN2003
Relays — Coil 5 V, Contacts 24 VAC 1A SPST (Minimum)	Radio Shack 275-243 or Other
Op-Amp — For Precision Temp Sensor Circuits QUAD Op-Amp	Motorola — Linear LM324
Precision Temperature Sensor — TO-92 Pkg	National Semiconductor LM34C

NOTE: This is only a partial parts list. Parts commonly found in lab stock are not shown.

4.3.2 Project Programming

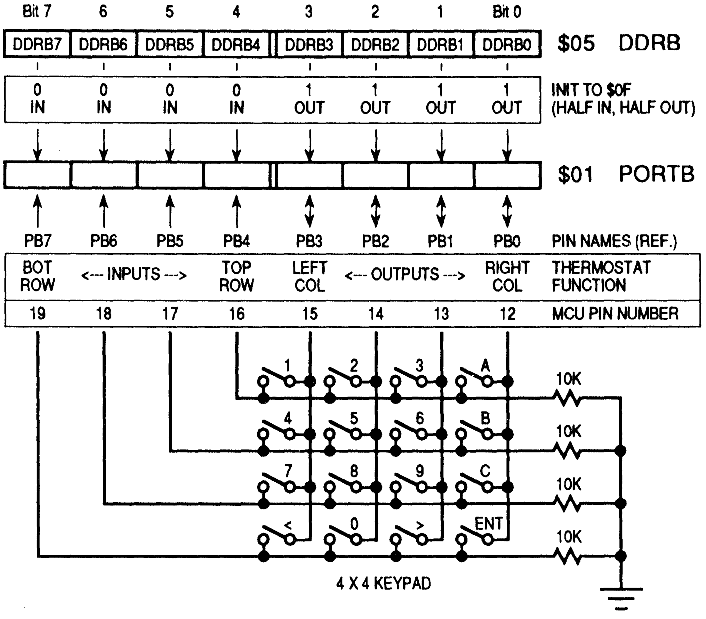
Figures 4-3, 4-4, 4-5, and 4-6 (MCU port summary information) act as a handy reference to the software programmer in the thermostat project. These figures summarize the most important information needed by the programmer.



Bit 7	6	5	4	3	2	1	Bit 0	
DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0	$04 DDRA
1 OUT	1 OUT	1 OUT	1 OUT	1 OUT	1 OUT	1 OUT	1 OUT	INIT TO $FF (ALL OUTPUTS)
								$00 PORTA
PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PIN NAMES (REF.)
LCD DATA 7	LCD DATA 6	LCD DATA 5	LCD DATA 4	LCD DATA 3	LCD DATA 2	LCD DATA 1	LCD DATA 0	THERMOSTAT FUNCTION
4	5	6	7	8	9	10	11	MCU PIN NUMBER
14	13	12	11	10	9	8	7	LCD PIN NUMBER

SEE PORT C FOR LCD SIGNALS - E, RS, AND R/W

Figure 4-3. Port A Summary



Bit 7	6	5	4	3	2	1	Bit 0	
DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	$05 DDRB
0 IN	0 IN	0 IN	0 IN	1 OUT	1 OUT	1 OUT	1 OUT	INIT TO $0F (HALF IN, HALF OUT)
								$01 PORTB
PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	PIN NAMES (REF.)
BOT ROW	<--- INPUTS --->		TOP ROW	LEFT COL	<--- OUTPUTS --->		RIGHT COL	THERMOSTAT FUNCTION
19	18	17	16	15	14	13	12	MCU PIN NUMBER

Figure 4-4. Port B Summary

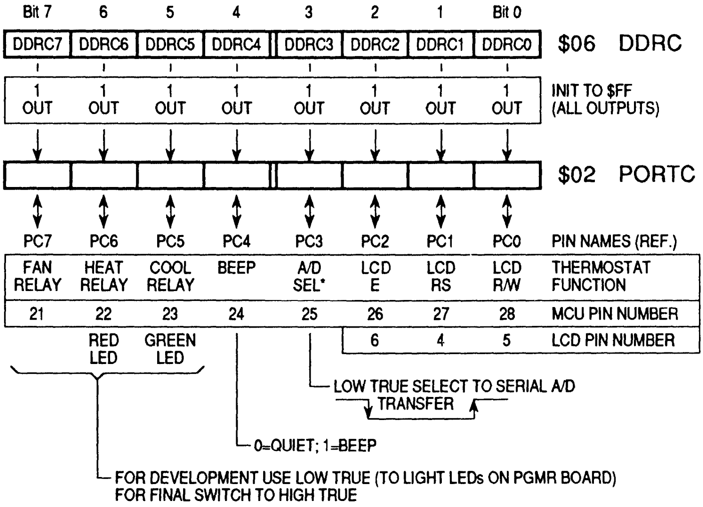

Bit 7 6 5 4 3 2 1 Bit 0
DDRC7 DDRC6 DDRC5 DDRC4 DDRC3 DDRC2 DDRC1 DDRC0
$06 DDRC
1 OUT 1 OUT 1 OUT 1 OUT 1 OUT 1 OUT 1 OUT 1 OUT
INIT TO $FF (ALL OUTPUTS)
$02 PORTC
PC7 PC6 PC5 PC4 PC3 PC2 PC1 PC0
PIN NAMES (REF.)
FAN RELAY
HEAT RELAY
COOL RELAY
BEEP
A/D SEL*
LCD E
LCD RS
LCD R/W
THERMOSTAT FUNCTION
21 22 23 24 25 26 27 28
MCU PIN NUMBER
RED LED
GREEN LED
6 4 5
LCD PIN NUMBER
LOW TRUE SELECT TO SERIAL A/D TRANSFER
0=QUIET; 1=BEEP
FOR DEVELOPMENT USE LOW TRUE (TO LIGHT LEDs ON PGMR BOARD)
FOR FINAL SWITCH TO HIGH TRUE

Figure 4-5. Port C Summary

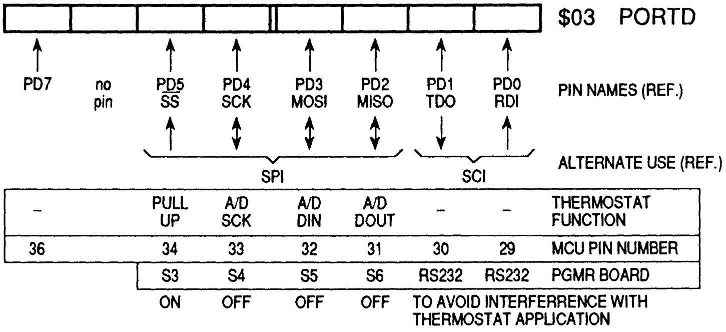


Figure 4-6. Port D Summary

After selecting major components and completing a preliminary hardware design, plan and begin writing software programs. You first write small programs that exercise the basic parts of the project. This procedure will expose any problems in the hardware design and will help you learn details of controlling major external peripherals.

Begin your project with a very simple program such as that shown in the assembler listing of **Figure 2-8 Assembler Listing**. You can easily modify the program to suit the keypad switches rather than wiring a switch as called for in the Figure 2-8 example. Also, you can modify the program to control the beeper rather than the red LED.

This first small program is meant to be very simple because you want to perform a crude check of the setup, as opposed to testing your programming ability. The simple example is not likely to have any significant programming problems.

Next, write a short program to check the LCD display. It is important to understand the operation of major elements, such as this display, before attempting a large program. Since there are so many possible causes of complete failure in a large program, you will have difficulty determining the source of your problems.

Figure 4-7 is a flowchart of the display checkout program. Figure 4-8 is the listing for this small program. Two subroutines (WCTRL and WDAT) were written to simplify operations with the LCD display. These subroutines will eventually become part of the final application program.

When this thermostat project was developed, the programs were not correct at first because the data sheet for the LCD display module was imprecise. The purpose of the small checkout programs is to work out these minor problems before beginning the large application program.

Application example programs shown in this applications guide can be tried in an MC68HC705C8 in one of two ways, depending upon their size.

For small programs (less than 176 bytes), you can download the example program to RAM (in the area $0051–$00FF) and execute it without programming any EPROM (so you don't have to erase EPROM to try another). To use this method, you must ORG your program at $0050 and the first byte **must** be the size of your example. The following procedure will provide the needed size byte.

1. Replace your ORG statement with the following two lines . . .

```
      ORG  $50
START FCB  END-START
```

2. After the last line in your program put . . .

```
END   EQU  *
```

3. Assemble the example program and make sure it ends at or before $00FF.

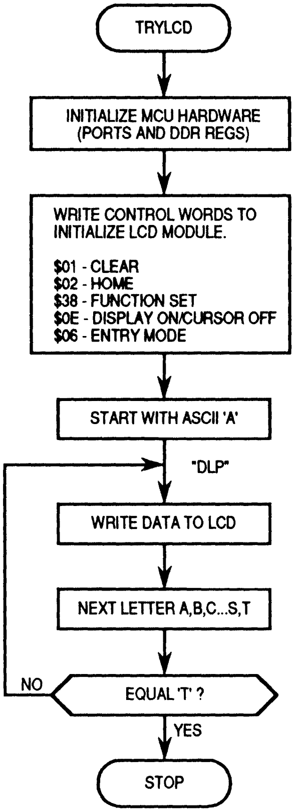


Figure 4-7. Display Checkout Flowchart

If the example program is too large to fit in the 176 bytes of RAM ($0050 to $00FF), you will have to program the example into EPROM and provide a reset vector. To provide a reset vector for a program example that begins with the label "BEGIN", put the following two lines at the end of your program:

```
ORG    $1FFE
FDB    BEGIN
```

NOTE

The example programs provided do not include a size byte or a reset vector; you will have to add whichever is appropriate for your situation.

```
              *********************************************************
              * TRYLCD - LCD Check out program                        *
              *         Initialize LCD module and display ABCDEF...S *
              *********************************************************

              * Register Equates
0000          PORTA   EQU   $00      LCD display data
0001          PORTB   EQU   $01      Keypad Row4,3,2,1;Col1,2,3,4
0002          PORTC   EQU   $02      Fan*,Heat*,Cool*,Beep;ADen*,E,RS,R/W
0004          DDRA    EQU   $04      Data direction, Port A (all output)
0005          DDRB    EQU   $05      Direction, Port B (7-4in,3-0out)
0006          DDRC    EQU   $06      Data direction, Port C (all output)

              * RAM Equates
009e          TEMPA   EQU   $9E      One byte temp storage location
009f          TEMPX   EQU   $9F      One byte temp storage location

0100                  ORG   $100
              * Set Port data patterns and directions
0100 a6 e8    TRYLCD  LDA   #$E8     Fan*,Heat*,Cool*,Beep;ADen*,E,RS,R/W
0102 b7 02            STA   PORTC    Initial Thermostat control values
0104 a6 ff            LDA   #$FF
0106 b7 04            STA   DDRA     Port A all outputs
0108 b7 06            STA   DDRC     Port C all outputs

              * LCD display peripheral needs to be initialized
010a a6 01            LDA   #$01
010c cd 01 2f         JSR   WCTRL    Clear
010f a6 02            LDA   #$02
0111 cd 01 2f         JSR   WCTRL    Home
0114 a6 38            LDA   #$38
0116 cd 01 2f         JSR   WCTRL    Function Set- 8-bit,2-line,5X7
0119 a6 0c            LDA   #$0C
011b cd 01 2f         JSR   WCTRL    Display on, Cursor off
011e a6 06            LDA   #$06
0120 cd 01 2f         JSR   WCTRL    Entry mode- Inc addr, no shift

0123 a6 41            LDA   #'A      ASCII 'A'
0125 cd 01 49 DLP     JSR   WDAT     Display a character
0128 4c               INCA           To next ASCII character
0129 a1 54            CMP   #'T      Go ABCDEFGHIJKLMNOPQRS & stop
012b 26 f8            BNE   DLP      Loop till T
012d 20 fe    HERE    BRA   HERE     Stop
```

Figure 4-8. Display Checkout Program Listing (Sheet 1 of 2)

```
                  *******************************************************
                  * WCTRL - Write control word to LCD peripheral        *
                  *       Enter with control word in accumulator        *
                  *       Return with original value of X               *
                  *       Delay ~4.5mS if A=$01 or $02 else delay ~120µS *
                  *******************************************************
012f bf 9f        WCTRL    STX   TEMPX     Save X
0131 b7 00                 STA   PORTA     Write control word to LCD
0133 14 02                 BSET  2,PORTC   E->1
0135 15 02                 BCLR  2,PORTC   E->0
0137 ae 14                 LDX   #20       20*6~*1µS/~=120µS
0139 5a           L120U    DECX            Delay loop ~120µS
013a 26 fd                 BNE   L120U     20-19,19-18...1-0
013c a1 02                 CMP   #$02      Commands $01 & $02 req extra delay
013e 22 06                 BHI   ARN5M     If command > $02 skip long delay
0140 cd 01 48     L5M      JSR   ANRTS     JSR+RTS TAKES 12~ (just want delay)
0143 5a                    DECX            TAKES 3~ (X=0->1 on first pass)
0144 26 fa                 BNE   L5M       3~ Loop 256*18~*1µS/~=4.608mS Delay
0146 be 9f        ARN5M    LDX   TEMPX     Restore X
0148 81           ANRTS    RTS             ** RETURN **

                  *******************************************************
                  * WDAT - Write data word to LCD peripheral            *
                  *       Enter with data word in accumulator           *
                  *       Return with original values of X & A          *
                  *       Delay ~120µS after data write                 *
                  *******************************************************
0149 bf 9f        WDAT     STX   TEMPX     Save X
014b b7 9e                 STA   TEMPA     Save A
014d b7 00                 STA   PORTA     Write data word to LCD
014f 12 02                 BSET  1,PORTC   RS->1
0151 14 02                 BSET  2,PORTC   E->1
0153 15 02                 BCLR  2,PORTC   E->0
0155 13 02                 BCLR  1,PORTC   RS->0
0157 ae 14                 LDX   #20       20*6~*1µS/~=120µS
0159 5a           L120     DECX            Delay loop ~120µS
015a 26 fd                 BNE   L120      20-19,19-18...1-0
015c b6 9e                 LDA   TEMPA     Restore A
015e be 9f                 LDX   TEMPX     Restore X
0160 81                    RTS             ** RETURN **
```

Figure 4-8. Display Checkout Program Listing (Sheet 2 of 2)

Since we now understand the LCD display, we can use the display to check out the keypad interface. To read a keypad key, we must recognize a key closure, delay to allow debounce, and decode the position (row/column) of the key. This is an example of how the MCU can simplify the hardware design. Software can be used to debounce the keys rather using complicated hardware circuits. Software also allows many switches to be wired in a row/column matrix so fewer I/O lines are needed.

The flowchart in Figure 4-9 shows how keypad keys are detected. Figure 4-10 is a listing of the keypad checkout program.

A real-time loop structure was chosen for the thermostat project main program. This basic structure can be used for many applications. The timing of the main loop determines the delays between activities in the complete application program.

A real time-of-day clock can easily be developed using the main loop time and simple software counters. Figure 4-11 is the flowchart for this basic loop structure. The complete listing for the thermostat project is included at the end of this section.

After a reset, there are a series of instructions to initialize ports, peripheral systems, and software variables. After this initialization, the main loop is entered and repeated continuously as long as power is applied.

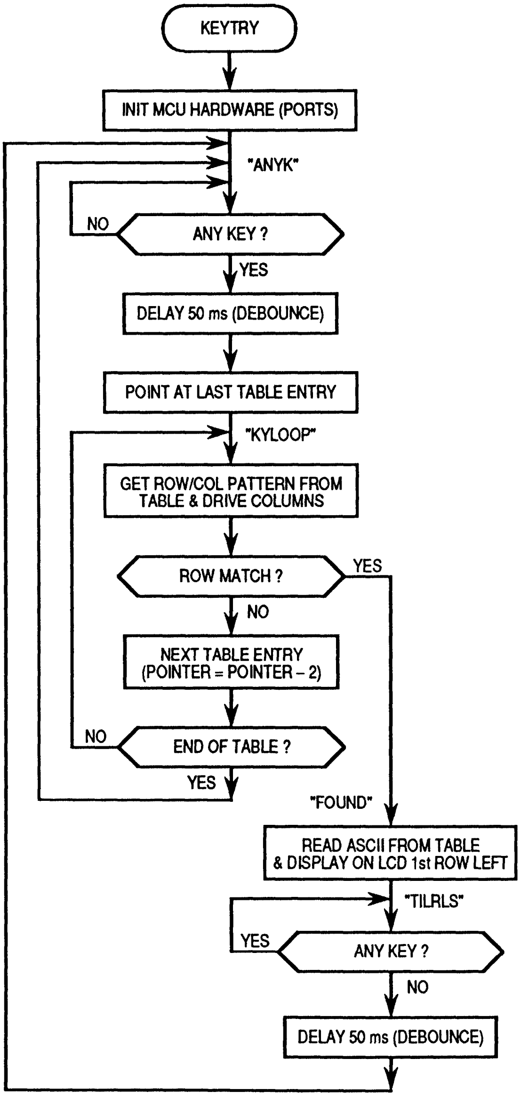

KEYTRY
INIT MCU HARDWARE (PORTS)
"ANYK"
NO
ANY KEY ?
YES
DELAY 50 ms (DEBOUNCE)
POINT AT LAST TABLE ENTRY
"KYLOOP"
GET ROW/COL PATTERN FROM TABLE & DRIVE COLUMNS
ROW MATCH ?
YES
NO
NEXT TABLE ENTRY (POINTER = POINTER – 2)
NO
END OF TABLE ?
YES
"FOUND"
READ ASCII FROM TABLE & DISPLAY ON LCD 1st ROW LEFT
"TILRLS"
YES
ANY KEY ?
NO
DELAY 50 ms (DEBOUNCE)

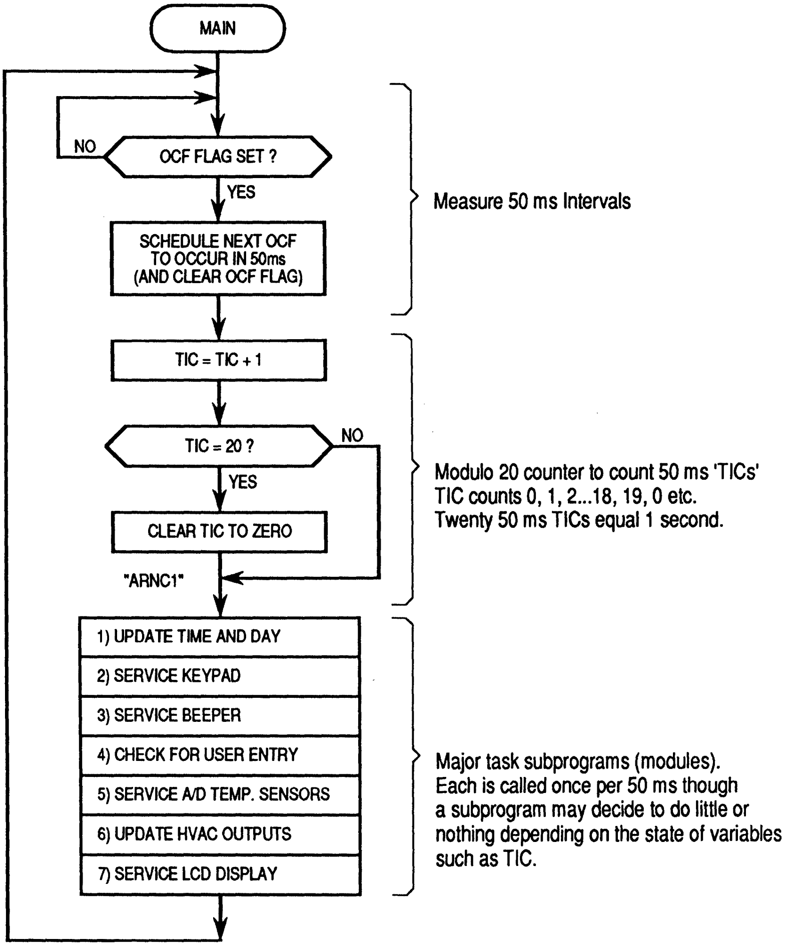
Figure 4-9. Keypad Checkout Flowchart

```
                    *********************************************************
                    * KEYTRY - Try out keypad debounce and decode software  *
                    *          Detect and debounce keys. When a key found   *
                    *          change it to ASCII and display on LCD        *
                    *          Debounce release of key and look for more    *
                    *********************************************************

                    * Register Equates
0000                PORTA    EQU   $00      LCD display data
0001                PORTB    EQU   $01      Keypad Row4,3,2,1;Col1,2,3,4
0002                PORTC    EQU   $02      Fan*,Heat*,Cool*,Beep;ADen*,E,RS,R/W
0004                DDRA     EQU   $04      Data direction, Port A (all output)
0005                DDRB     EQU   $05      Direction, Port B (7-4in,3-0out)
0006                DDRC     EQU   $06      Data direction, Port C (all output)

                    * RAM Equates
009d                KEYVAL   EQU   $9D      Keypad key (ASCII)
009e                TEMPA    EQU   $9E      One byte temp storage location
009f                TEMPX    EQU   $9F      One byte temp storage location

0100                         ORG   $100
                    * Set Port data patterns and directions
0100 a6 e8          INIT     LDA   #$E8     Fan*,Heat*,Cool*,Beep;ADen*,E,RS,R/W
0102 b7 02                   STA   PORTC    Initial Thermostat control values
0104 4f                      CLRA           Row3,2,1,0;Col1,2,3,4
0105 b7 01                   STA   PORTB    All cols initially off
0107 4a                      DECA           to $FF
0108 b7 04                   STA   DDRA     Port A all outputs
010a b7 06                   STA   DDRC     Port C all outputs
010c a6 0f                   LDA   #$0F     Rows=in, Cols=outs
010e b7 05                   STA   DDRB     Port B half ins, half outs

                    * LCD display peripheral needs to be initialized
0110 a6 01                   LDA   #$01
0112 cd 01 93                JSR   WCTRL    Clear
0115 a6 02                   LDA   #$02
0117 cd 01 93                JSR   WCTRL    Home
011a a6 38                   LDA   #$38
011c cd 01 93                JSR   WCTRL    Function Set- 8-bit,2-line,5X7
011f a6 0c                   LDA   #$0C
0121 cd 01 93                JSR   WCTRL    Display on, Cursor off
0124 a6 06                   LDA   #$06
0126 cd 01 93                JSR   WCTRL    Entry mode- Inc addr, no shift

                    ** END of INITIALIZATION ********************************
```

Figure 4-10. Keypad Checkout Program Listing (Sheet 1 of 2)

```
0129 a6 0f    KEYTRY  LDA   #$0F
012b b7 01            STA   PORTB    Turn on all cols
012d b6 01    ANYK    LDA   PORTB    Reads rows in upper 4
012f a4 f0            AND   #$F0     Mask away cols
0131 27 fa            BEQ   ANYK     Loop till a key is found
0133 cd 01 65         JSR   DLY50    Debounce key
0136 ae 1e            LDX   #30      Pointer to last pair in KYTBL
0138 d6 01 73 KYLOOP  LDA   KYTBL,X  Get row/col pattern
013b b7 01            STA   PORTB    Drive cols
013d b1 01            CMP   PORTB    Check for row & col match
013f 27 06            BEQ   FOUND    If =; key found
0141 5a               DECX           Point to next pair of entries
0142 5a               DECX           in KYTBL
0143 2a f3            BPL   KYLOOP   Loop if more entries
0145 20 e2            BRA   KEYTRY   Key gone; start over
0147 d6 01 74 FOUND   LDA   KYTBL+1,X  Get key equiv from table
014a b7 9d            STA   KEYVAL   Save for now
014c a6 80            LDA   #$80     Left end of 1st row
014e cd 01 93         JSR   WCTRL    Position entry point
0151 b6 9d            LDA   KEYVAL   Get the ASCII key value
0153 cd 01 ad         JSR   WDAT     Display the key
0156 a6 0f            LDA   #$0F
0158 b7 01            STA   PORTB    Turn on all cols
015a b6 01    TILRLS  LDA   PORTB    Reads rows in upper 4
015c a4 f0            AND   #$F0     Mask away cols
015e 26 fa            BNE   TILRLS   Loop till no key pressed
0160 cd 01 65         JSR   DLY50    Debounce release
0163 20 c4            BRA   KEYTRY   Look for another key
              *********************************
              * Keypad Correspondance Table
              *  1st entry of each pair is Row/Col bit pattern
              *  2nd entry of each pair is ASCII equiv of key
              * COL # ->   1  2  3  4
              *            v  v  v  v
              * ROW 1 ->   1  2  3  A
              * ROW 2 ->   4  5  6  B
              * ROW 3 ->   7  8  9  C
              * ROW 4 ->   <  0  >  !
0173 18 31    KYTBL   FCB   $18,'1   Row 1, Col 1 (Top Left)
0175 28 34            FCB   $28,'4   Row 2, Col 1
0177 48 37            FCB   $48,'7   Row 3, Col 1
0179 88 3c            FCB   $88,'<   Row 4, Col 1
017b 14 32            FCB   $14,'2   Row 1, Col 2
017d 24 35            FCB   $24,'5   Row 2, Col 2
017f 44 38            FCB   $44,'8   Row 3, Col 2
0181 84 30            FCB   $84,'0   Row 4, Col 2
0183 12 33            FCB   $12,'3   Row 1, Col 3
0185 22 36            FCB   $22,'6   Row 2, Col 3
0187 42 39            FCB   $42,'9   Row 3, Col 3
0189 82 3e            FCB   $82,'>   Row 4, Col 3
018b 11 41            FCB   $11,'A   Row 1, Col 4
018d 21 42            FCB   $21,'B   Row 2, Col 4
018f 41 43            FCB   $41,'C   Row 3, Col 4
0191 81 21            FCB   $81,'!   Row 4, Col 4 (Bot Right)
```

Figure 4-10. Keypad Checkout Program Listing (Sheet 2 of 2)



Measure 50 ms Intervals

Modulo 20 counter to count 50 ms 'TICs'
TIC counts 0, 1, 2...18, 19, 0 etc.
Twenty 50 ms TICs equal 1 second.

Major task subprograms (modules).
Each is called once per 50 ms though
a subprogram may decide to do little or
nothing depending on the state of variables
such as TIC.

Figure 4-11. Main Program Flowchart

```
                  *********************************************
                  * MC68HC705C8 Example Development Project   *
                  *   A Home Thermostat with indoor/outdoor   *
                  *   temperature and time-of-day             *
                  *                                           *
                  *   This example uses an LCD display, a 4x4 *
                  *   keypad, a piezo beeper, and an MC145041 *
                  *   serial A/D converter.                   *
                  *                                           *
                  *   Software is configured in a real-time   *
                  *   loop and demonstrates timing techniques *
                  *   and program modularity principles.      *
                  *                                           *
                  *   The project is complete enough to show  *
                  *   the development process but is not      *
                  *   intended to be a finished product.      *
                  *********************************************

                  * Register Equates
0000              PORTA    EQU   $00      LCD display data
0001              PORTB    EQU   $01      Keypad Row4,3,2,1;Col1,2,3,4
0002              PORTC    EQU   $02      Fan*,Heat*,Cool*,Beep;ADen*,E,RS,R/W
0003              PORTD    EQU   $03      in,-,SS*,SCK;MOSI,MISO,TxD,RxD
0004              DDRA     EQU   $04      Data direction, Port A (all output)
0005              DDRB     EQU   $05      Data direction, Port B (7-4in,3-0out)
0006              DDRC     EQU   $06      Data direction, Port C (all output)
000a              SPCR     EQU   $0A      SPIE,SPE,-,MSTR;CPOL,CPHA,SPR1,SPR0
000b              SPSR     EQU   $0B      SPIF,WCOL,-,MODF;-,-,-,-
000c              SPDR     EQU   $0C      SPI Data
000d              BAUD     EQU   $0D      -,-,SCP1,SCP0;-,SCR2,SCR1,SCR0
000e              SCCR1    EQU   $0E      R8,T8,-,M;WAKE,-,-,-
000f              SCCR2    EQU   $0F      TIE,TCIE,RIE,ILIE;TE,RE,RWU,SBK
0010              SCSR     EQU   $10      TDRE,TC,RDRF,IDLE;OR,NF,FE,-
0011              SCDR     EQU   $11      SCI Data
0011              RDR      EQU   $11      SCI Receive Data (same as SCDR)
0011              TDR      EQU   $11      SCI Transmit Data (same as SCDR)
0012              TCR      EQU   $12      ICIE,OCIE,TOIE,0;0,0,IEGE,OLVL
0013              TSR      EQU   $13      ICF,OCF,TOF,0; 0,0,0,0
0014              ICAP     EQU   $14      Input Capture Reg (Hi-$14, Lo-$15)
0016              OCMP     EQU   $16      Output Compare Reg (Hi-$16, Lo-$17)
0018              TCNT     EQU   $18      Timer Count Reg (Hi-$18, Lo-$19)
001a              ALTCNT   EQU   $1A      Alternate Count Reg (Hi-$1A, Lo-$1B)

                  * RAM Equates
00a0                       ORG   $A0
                  * Using 'A6 to debug and monitor uses lower RAM

00a0              TEMPA    RMB   1        One byte temp storage location
00a1              TEMPX    RMB   1        One byte temp storage location
00a2              TIC      RMB   1        50mS Tics 00-19 20 Tics = 1 Sec
00a3              SEC      RMB   1        Current Time Seconds 00-59
```

```
00a4          BCDEQ    RMB   1         BCD equivalent of ENTRY
              * it's easier to roll in new digits to a BCD buffer vs binary.

              * Next 7 entries are accessed by indexed addressing
              * using a 1 byte
              * offset from ENTRY. The offset is MODE (in X) and the value at
              * ENTRY,X is the value that is subject to change in the selected
mode.
00a5          ENTRY    RMB   1         Binary value being entered by user
00a6          HR       RMB   1         Current Time Hour 1-12 (binary)
00a7          MIN      RMB   1         Current Time Minute 00-59 (binary)
00a8          AMPM     RMB   1         Current Time AM=0, PM=1
00a9          DAY      RMB   1         Day of Wk 1=Sun...7=Sat
00aa          HVACM    RMB   1         HVAC Equipment Mode
              * Modes 0 - Off
              *       1 - Heat
              *       2 - Cool
              *       3 - Fan Only

00ab          GOAL     RMB   1         Goal temp. setting (+)
              * End of values accessed by offset from ENTRY

00ac          INTMP    RMB   1         Current Indoor Temperature (+)
00ad          OUTMP    RMB   1         Current Outdoor Temperature (+/-)

00ae          ASC100   RMB   1         ASCII hundreds digit (-,<sp>,1, or 2)
00af          ASC10    RMB   1         ASCII tens digit (0 thru 9)
00b0          ASC1     RMB   1         ASCII ones digit (0 thru 9)

00b1          MODE     RMB   1         Current Mode (for user interfce)
              * Modes 0 - Inactive; display shows current time/temp/etc.
              *       1 - Set Time HR
              *       2 - Set Time MIN
              *       3 - Set Time AM/PM
              *       4 - Set Time DAY
              *       5 - Set HVAC Mode - Off, Heat, Cool, Fan Only
              *       6 - Set Target Temperature

00b2          HVACON   RMB   1         0=off, 1=on (running now)
00b3          KEYVAL   RMB   1         Keypad key (ASCII) or debounce state
00b4          BEEPM    RMB   1         Beeper request
              * 2=>single 100mS beep, 8=>double beep, 26=>1 sec beep

00b5          ACTIMR   RMB   1         Activity timer
              * Set=60 sec on key, decrement 1/sec, if 0 mode reverts to 0

00b6          ENTFLG   RMB   1         New entry flag, 0-new 1-old
              * Entries default to current value when new. If user enters
              * a single digit the tens digit is cleared. If user enters
              * more digits they shift in from rt. so new digit is 1's, old
              * 1's becomes 10's, and old 10's falls off left (lost).
```

```
0100                ORG   $0100    Program will start at $0100
             * $0100 is the start of EPROM in the '705C8

             * Initialization done at reset & on detection of some errors

0100 9c      INIT   RSP            Reset stack pointer to $FF

             * Set Port data patterns and directions
0101 a6 e8          LDA   #$E8     Fan*,Heat*,Cool*,Beep;ADen*,E,RS,R/W
0103 b7 02          STA   PORTC    Initial values for Thermostat controls
0105 4f             CLRA           Row3,2,1,0;Col1,2,3,4
0106 b7 01          STA   PORTB    All cols initially off
0108 4a             DECA           to $FF
0109 b7 04          STA   DDRA     Port A all outputs
010b b7 06          STA   DDRC     Port C all outputs
010d a6 0f          LDA   #$0F     Rows=in, Cols=outs
010f b7 05          STA   DDRB     Port B half ins, half outs

             * Set up SPI to talk to ext serial A/D converter MC145041

             **
             ** CAUTION !! S3 thru S6 on PGMR Board can conflict with SPI
             **
0111 b6 03   WAITSW LDA   PORTD    Wait 'till S3-on, S4, S5, S6-off
0113 a4 3C          AND   #$3C     only care about S3,thru S6
0115 a1 20          CMP   #$20     S3-on, S4, S5, S6-off ?
0117 26 f8          BNE   WAITSW   If not wait till they are
             * Previous 4 lines only needed for development on PGMR board

0119 a6 50          LDA   #$50     SPIE,SPE,-,MSTR;CPOL,CPHA,SPR1,SPR0
011b b7 0a          STA   SPCR     SPI on as Master, 2µS norm low clock

             * SCI not used in this application
             * Timer output compare used to time 50mS loop
011d 4f             CLRA           ICIE,OCIE,TOIE,0;0,0,IEGE,OLVL
011e b7 12          STA   TCR      no timer interrupts or pins used

             * LCD display peripheral needs to be initialized
0120 a6 01          LDA   #$01
0122 cd 06 20       JSR   WCTRL    Clear
0125 a6 02          LDA   #$02
0127 cd 06 20       JSR   WCTRL    Home
012a a6 38          LDA   #$38
012c cd 06 20       JSR   WCTRL    Function Set- 8-bit,2-line,5X7
012f a6 0c          LDA   #$0C
0131 cd 06 20       JSR   WCTRL    Display on, Cursor off
0134 a6 06          LDA   #$06
0136 cd 06 20       JSR   WCTRL    Entry mode- Inc addr, no shift
```

```
                * Set time to 12:00 AM SUN
0139 3f a2              CLR   TIC       Init 50mS counter
013b 3f a3              CLR   SEC       Init seconds to 0
013d a6 0c              LDA   #12       Hr=12
013f b7 a6              STA   HR
0141 3f a7              CLR   MIN       Min=00
0143 3f a8              CLR   AMPM      AM (AMPM=0)
0145 a6 01              LDA   #1        Sun-1,Sat-7
0147 b7 a9              STA   DAY       Day=Sunday

0149 3f b1              CLR   MODE      Set user interface to inactive
014b 3f b3              CLR   KEYVAL    Say no key closed
014d 3f b4              CLR   BEEPM     Set beeper request to off
014f 3f b2              CLR   HVACON    Indicate HVAC Equip not running now
0151 3f aa              CLR   HVACM     Set HVAC Equip mode to off
0153 a6 48              LDA   #72
0155 b7 ab              STA   GOAL      Set default goal temp to 72°F

                ** END of INITIALIZATION ********************************
```

```
              *******************************************************
              * MAIN - Beginning of main program loop               *
              *      Loop is executed once every 50mS (exactly)     *
              *      A pass through all major task routines takes   *
              *      less than 50mS and then time is wasted until   *
              *      the output compare flag gets set (every 50mS). *
              *      When an output compare triggers, the flag is   *
              *      cleared & 12500 is added to the compare reg    *
              *      so the next trigger will occur in exactly 50mS *
              *      (12500*4µS/cnt = 50mS).(Xtal = 2MHz, bus = 1MHz)*
              *                                                     *
              *      The variable TIC keeps track of 50mS periods   *
              *      when TIC increments from 19 to 20 it is cleared *
              *      to 0 and seconds are incremented.              *
              *                                                     *
              *      The keypad is checked every 50mS pass and a new *
              *      closure or release is not acted upon until the *
              *      pass after it is first seen.  This acts as a   *
              *      switch debounce.                               *
              *                                                     *
              *      The display is updated only when seconds change.*
              *      Display call is at bottom of main loop so any  *
              *      change caused by a key is reflected in the     *
              *      display update.                                *
              *                                                     *
              *      Temperature readings are only taken once/sec   *
              *******************************************************
0157 0d 13 fd MAIN     BRCLR 6,TSR,MAIN Loop here till OCF flag set
015a b6 17             LDA   OCMP+1   Low byte of OC register
015c ab d4             ADD   #$D4     Low half of 12500
015e b7 a0             STA   TEMPA    Save till high half calculated
0160 b6 16             LDA   OCMP     High byte of OC register
0162 a9 30             ADC   #$30     High half of 12500 (+carry)
0164 b7 16             STA   OCMP     Update OC reg
0166 b6 a0             LDA   TEMPA    Get low half of updated value
0168 b7 17             STA   OCMP+1   Update low half of OC reg
              * OC now = old OC + 12500, and OCF flag is clear
016a b6 a2             LDA   TIC      Get current TIC value
016c 4c                INCA           TIC=TIC+1
016d b7 a2             STA   TIC      Update TIC
016f a1 14             CMP   #20      20th TIC ?
0171 25 02             BLO   ARNC1    If not, skip next clear
0173 3f a2             CLR   TIC      Clear TIC on 20th
              * End of synchronization to 50mS TIC; Run main tasks and
              *  branch back to main within 50mS.  Sync OK as long as
              *  no 2 consecutive passes take more than 100mS.

0175 cd 01 8c ARNC1    JSR   TIME     Update time-of-day & day-of-week
0178 cd 01 c9          JSR   KYPAD    Check/service keypad
017b cd 02 16          JSR   BEEP     Update Beeper
017e cd 02 2f          JSR   USER     User Interface to set time, temp, etc.
0181 cd 03 09          JSR   A2D      Check Temp Sensors
0184 cd 03 34          JSR   HVAC     Update Heat/Air Cond Outputs
0187 cd 03 9d          JSR   LCD      Update LCD display
018a 20 cb             BRA   MAIN     Back to Top & wait for next TIC
              ** END of Main Loop ************************************
```

```
                *********************************************************
                * TIME - Update Time-of-day & Day-of-week               *
                *  If TIC not = 0, just skip whole routine              *
                *       When SEC rolls 59->0, inc MIN                   *
                *       When MIN rolls 59->0, inc HR                    *
                *       When HR rolls 11->12, change AMPM 1->0 or 0->1  *
                *       When AMPM chgs 1->0, inc DAY                    *
                *       When DAY rolls 7->8, set to 1 (Sun)             *
                *********************************************************
018c            TIME     EQU   *          Update Time-of-day & Day-of-week
018c 3d a2               TST   TIC        Check for TIC=zero
018e 26 38               BNE   XTIME      If not; just exit
0190 3c a3               INC   SEC        SEC=SEC+1
0192 a6 3c               LDA   #60
0194 b1 a3               CMP   SEC        Did SEC -> 60 ?
0196 26 30               BNE   XTIME      If not; just exit
0198 3f a3               CLR   SEC        Seconds rollover
019a 3c a7               INC   MIN        MIN=MIN+1
019c b1 a7               CMP   MIN        A still 60; MIN=60 ?
019e 26 28               BNE   XTIME      If not; just exit
01a0 3f a7               CLR   MIN        Minutes rollover
01a2 3c a6               INC   HR         HR=HR+1
01a4 b6 a6               LDA   HR         For comparisons
01a6 a1 0d               CMP   #13        HR=13 ?
01a8 26 06               BNE   ARNS1      If not; skip
01aa a6 01               LDA   #1
01ac b7 a6               STA   HR         Set HR=1
01ae 20 18               BRA   XTIME      Exit
01b0 a1 0c      ARNS1    CMP   #12        HR=12 ?
01b2 26 14               BNE   XTIME      If not; just exit
01b4 b6 a8               LDA   AMPM
01b6 a8 01               EOR   #%00000001  Invert AM/PM bit
01b8 b7 a8               STA   AMPM       0=AM, 1=PM
01ba 26 0c               BNE   XTIME      If not AM now; just exit
01bc 3c a9               INC   DAY        DAY=DAY+1
01be b6 a9               LDA   DAY
01c0 a1 08               CMP   #8         Day rollover ?
01c2 26 04               BNE   XTIME      If not; just exit
01c4 a6 01               LDA   #1
01c6 b7 a9               STA   DAY        Set Day to 1 (SUN)
01c8 81         XTIME    RTS   ** RETURN from TIME **
```

```
                   *******************************************************
                   * KYPAD - Check for & decode keys                     *
                   *       KEYVAL indicates ASCII equivalent of key or   *
                   *       debounce status as follows                    *
                   *  $00 - no key pressed, look for any closure         *
                   *  $01 - key closed 50mS ago (debounce), decode now   *
                   *  $20-$7F - key found, debounced, & decoded (not seen) *
                   *  $FE - key recognized by some task, wait for release *
                   *  $FF - key released 50mS ago (debounce release)     *
                   *******************************************************
01c9               KYPAD    EQU    *         Check for & decode keys
01c9 b6 b3                  LDA    KEYVAL    KEYVAL indicates what to do
01cb 26 0e                  BNE    CHK401    If not 0; Check for $01
01cd a6 0f                  LDA    #$0F
01cf b7 01                  STA    PORTB     Turn on all cols
01d1 b6 01                  LDA    PORTB     Reads rows in upper 4
01d3 a4 f0                  AND    #$F0      Mask away cols
01d5 27 3e                  BEQ    XKYPAD    Exit if no key
01d7 3c b3                  INC    KEYVAL    To $01
01d9 20 3a                  BRA    XKYPAD    Exit, key will be decoded in 50mS
01db a1 01         CHK401   CMP    #$01      KEYVAL=$01 ?
01dd 26 1c                  BNE    CHK4FE    If not 0; Check for $FE
01df ae 1e                  LDX    #30       Pointer to last pair in KYTBL
01e1 d6 06 00 KYLOOP        LDA    KYTBL,X   Get row/col pattern
01e4 b7 01                  STA    PORTB     Drive cols
01e6 b1 01                  CMP    PORTB     Check for row & col match
01e8 27 06                  BEQ    FOUND     If =; key found
01ea 5a                     DECX             Point to next pair of entries
01eb 5a                     DECX             in KYTBL
01ec 2a f3                  BPL    KYLOOP    Loop if more entries
01ee 3f b3                  CLR    KEYVAL    No key found; set KEYVAL=0
01f0 d6 06 01 FOUND         LDA    KYTBL+1,X  Get key equiv from table
01f3 b7 b3                  STA    KEYVAL    $20 ≤ KEYVAL ≤ $7F
01f5 a6 02                  LDA    #2
01f7 b7 b4                  STA    BEEPM     Request beep as feedback
01f9 20 1a                  BRA    XKYPAD    Exit
01fb a1 fe         CHK4FE   CMP    #$FE      KEYVAL=$FE ?
01fd 26 10                  BNE    CHK4FF    If not check for $FF
01ff a6 0f                  LDA    #$0F
0201 b7 01                  STA    PORTB     Turn on all cols
0203 b6 01                  LDA    PORTB     Reads rows in upper 4
0205 a4 f0                  AND    #$F0      Mask away cols
0207 26 0c                  BNE    XKYPAD    Exit if key still closed
0209 a6 ff                  LDA    #$FF
020b b7 b3                  STA    KEYVAL    Set KEYVAL=$FF
020d 20 06                  BRA    XKYPAD    & Exit
020f a1 ff         CHK4FF   CMP    #$FF      KEYVAL=$FF ?
0211 26 02                  BNE    XKYPAD    If not; exit
0213 3f b3                  CLR    KEYVAL    Set KEYVAL=$00
0215 81            XKYPAD   RTS    ** RETURN from KYPAD **
```

```
                 *********************************************************
                 * BEEP - Update audible beeper                          *
                 *       Single 100mS beep on key closure (feedback)     *
                 *       Beep (100mS/on, 200off, 100on) entry accepted   *
                 *       Beep 1 second to indicate entry error           *
                 *********************************************************
0216             BEEP     EQU    *        Update audible beep
0216 b6 b4                LDA    BEEPM    BEEPM indicates what to do
0218 26 04                BNE    ACTIV    Branch if beeper active
021a 19 02                BCLR   4,PORTC  Turn off beeper
021c 20 10                BRA    XBEEP    & Exit

021e 3a b4       ACTIV    DEC    BEEPM    Times beeps
                 * Accumulator has undecremented version of BEEPM
                 * Beeper should be on unless BEEPM is between 3 and 6
0220 a1 03                CMP    #3
0222 25 08                BLO    BPRON    If <3 turn beeper on
0224 a1 06                CMP    #6
0226 22 04                BHI    BPRON    If >6 turn beeper on
0228 19 02                BCLR   4,PORTC  Turn beeper off
022a 20 02                BRA    XBEEP    & Exit
022c 18 02       BPRON    BSET   4,PORTC  Turn beeper on
022e 81          XBEEP    RTS    ** RETURN from BEEP **
```

```
                ******************************************************
                * USER - User Interface to set time, temp, etc.      *
                *  Variable named MODE identifies current user function *
                *  0 - Inactive; display shows current time/temp/etc.  *
                *  1 - Set Time HR                                    *
                *  2 - Set Time MIN                                   *
                *  3 - Set Time AM/PM                                 *
                *  4 - Set Time DAY                                   *
                *  5 - Set HVAC Mode - Off, Heat, Cool, Fan Only      *
                *  6 - Set Target Temperature                         *
                *       MODE reverts to 0-inactive if no keys for 1 min *
                *       To activate modes press A until desired value  *
                *       to be changed is blinking. Next enter desired  *
                *       setting numbers and press enter (!).           *
                * Current program does not use <,>,B, or C keys.      *
                ******************************************************
022f            USER    EQU   *         User Interface to set time, temp, etc.
022f 3d a3              TST   SEC       Seconds = 0 ?
0231 26 0a              BNE   CHKEY     If not, skip ACTIMR
0233 3a b5              DEC   ACTIMR    Decrement activity timer
0235 26 02              BNE   ARMCLR    No activity for 1 minute
0237 3f b1              CLR   MODE      Force to inactive
0239 2a 02      ARMCLR  BPL   CHKEY     Did ACTIMR roll neg ?
023b 3f b5              CLR   ACTIMR    If so clear it
023d b6 b3      CHKEY   LDA   KEYVAL    Get key value
023f a1 20              CMP   #$20      Ignore key if <$20 or >$7F
0241 25 04              BLO   XUSER2    Exit if <$20
0243 a1 7f              CMP   #$7F      ? > $7F is invalid
0245 23 03              BLS   VALKEY    Valid
0247 cc 02 ba   XUSER2  JMP   XUSER     May be too far to branch

                * Valid key has been detected
024a ae 3c      VALKEY  LDX   #60       60 seconds
024c bf b5              STX   ACTIMR    Set to timeout in 1 min.
024e a1 41              CMP   #'A       KEYVAL = A ?
0250 27 52              BEQ   NXTMOD    Advance to next setting
0252 a1 30              CMP   #'0       ASCII 0
0254 25 33              BLO   TRYENT    Branch if < 0
0256 a1 39              CMP   #'9       ASCII 9
0258 22 2f              BHI   TRYENT    BRANCH IF > 9
025a 3d b6              TST   ENTFLG    First # in entry ?
025c 26 06              BNE   NOFST     Skip if not
025e 3f a5              CLR   ENTRY     Clear ENTRY
0260 3f a4              CLR   BCDEQ     & its BCD equivalent
0262 3c b6              INC   ENTFLG    0->1 (NO LONGER 1st)
0264 48         NOFST   ASLA            Get hex 0-9 in left nibble
0265 48                 ASLA
0266 48                 ASLA
0267 48                 ASLA            nnnn 0000 & BCDEQ = xxxx yyyy
```

```
0268 48                 ASLA              Roll new digit into BCD
0269 39 a4              ROL    BCDEQ      Equiv of ENTRY
026b 48                 ASLA              With 4 double byte
026c 39 a4              ROL    BCDEQ      left shifts
026e 48                 ASLA
026f 39 a4              ROL    BCDEQ
0271 48                 ASLA
0272 39 a4              ROL    BCDEQ      BCDEQ now = yyyy nnnn
0274 b6 a4              LDA    BCDEQ
0276 a4 0f              AND    #$0F       Mask off 10's
0278 b7 a5              STA    ENTRY      Temp save 1's
027a b6 a4              LDA    BCDEQ      Get BCD again
027c 44                 LSRA              Right justify 10's
027d 44                 LSRA
027e 44                 LSRA
027f 44                 LSRA
0280 ae 0a              LDX    #10
0282 42                 MUL               A <- 10 * BCD 10's
0283 bb a5              ADD    ENTRY      Add in ones
0285 b7 a5              STA    ENTRY      Now binary equiv of BCDEQ
0287 20 2d              BRA    KEYFE      Acknowledge key and leave
0289 a1 21     TRYENT   CMP    #'!        Enter key ?
028b 26 29              BNE    KEYFE      If not, Ack key & leave
028d cd 02 bb           JSR    CHKPNT     Check for legal entry
               * On return N-bit indicates legal (Positive) & X points
               * at applicable value to be changed (HR,MIN,AMPM,DAY etc.)
0290 2a 0c              BPL    LEGENT     Branch if legal
0292 e6 a5              LDA    ENTRY,X    Get current value
0294 b7 a5              STA    ENTRY      Revert to current (legal) value
0296 3f b6              CLR    ENTFLG     So next # treated as first
0298 a6 1a              LDA    #26        26 * 50mS = 1.3 sec
029a b7 b4              STA    BEEPM      Beep 1S/200mS-off/100mS-on
029c 20 18              BRA    KEYFE      Acknowledge entry attempt
029e e7 a5     LEGENT   STA    ENTRY,X    Update value being set
02a0 a6 08              LDA    #8         100mS-on/200mS-off/100mS-on
02a2 b7 b4              STA    BEEPM      Double beep
02a4 3c b1     NXTMOD   INC    MODE       Adv to next setting
02a6 b6 b1              LDA    MODE       Check for past 6
02a8 a1 07              CMP    #7         <7?
02aa 25 02              BLO    NOCLR      If OK skip clear
02ac 3f b1              CLR    MODE       Rollover to 0
02ae be b1     NOCLR    LDX    MODE       Use as index to current
02b0 e6 a5              LDA    ENTRY,X    Get current value of entry
02b2 b7 a5              STA    ENTRY      Use current as default setting
02b4 3f b6              CLR    ENTFLG     Indicate next # is 1st
02b6 a6 fe     KEYFE    LDA    #$FE
02b8 b7 b3              STA    KEYVAL     Acknowledge key closures
02ba 81        XUSER    RTS    ** RETURN from USER **
```

```
                ***
                * CHKPNT - a utility subroutine used by USER routine
                *          Checks for entry within legal limits which
                *          depend on value being changed. HR=1-12, MIN=0-59
                *          and so on. If legal, N bit will be 0 (Positive).
                *          On return A has enrty value (or $FF if illegal)
                *          and X points at value to be changed. ENTRY,X
                *          may be used to access value to be changed.
02bb b6 a5      CHKPNT   LDA   ENTRY     For compares to chk limits
02bd be b1               LDX   MODE      For compares & as return pointer
02bf a3 01               CPX   #1        Set HR ?
02c1 26 08               BNE   TRI2      If not
02c3 a1 01               CMP   #1        <1?
02c5 25 04               BLO   TRI2      illegal (will ripple through)
02c7 a1 0c               CMP   #12       1-12 ?
02c9 23 3b               BLS   OKENT     Valid HR entry
02cb a3 02      TRI2     CPX   #2        Set MIN ?
02cd 26 07               BNE   TRI3      If not
02cf 4d                  TSTA            <0?
02d0 2b 04               BMI   TRI3      illegal (will ripple through)
02d2 a1 3b               CMP   #59       0-59 ?
02d4 23 30               BLS   OKENT     Valid MIN entry
02d6 a3 03      TRI3     CPX   #3        Set AMPM ?
02d8 26 07               BNE   TRI4      If not
02da 4d                  TSTA            <0?
02db 2b 04               BMI   TRI4      illegal (will ripple through)
02dd a1 01               CMP   #1        0 or 1 ?
02df 23 25               BLS   OKENT     Valid AMPM entry
02e1 a3 04      TRI4     CPX   #4        Set DAY ?
02e3 26 08               BNE   TRI5      If not
02e5 a1 01               CMP   #1        <1?
02e7 25 04               BLO   TRI5      illegal (will ripple through)
02e9 a1 07               CMP   #7        1-7 ?
02eb 23 19               BLS   OKENT     Valid DAY entry
02ed a3 05      TRI5     CPX   #5        Set HVAC Mode ?
02ef 26 07               BNE   TRI6      If not
02f1 4d                  TSTA            <0?
02f2 2b 04               BMI   TRI6      illegal (will ripple through)
02f4 a1 03               CMP   #3        0-3 ?
02f6 23 0e               BLS   OKENT     Valid HVACM entry
02f8 a3 06      TRI6     CPX   #6        Set GOAL Temp ?
02fa 26 08               BNE   BADENT    Illegal entry
02fc a1 32               CMP   #50       <50°F ?
02fe 25 04               BLO   BADENT    illegal
0300 a1 63               CMP   #99       < or = 99°F ?
0302 23 02               BLS   OKENT     Valid goal temp
0304 a6 ff      BADENT   LDA   #$FF      A negative value to set N
0306 b7 a5      OKENT    STA   ENTRY     Sets/or clears N
0308 81                  RTS             ** Return from CHKPNT **
                *  !!! There is more to this exit than is obvious. X=MODE
                * so X points at entry to be changed HR,MIN,AMPM,DAY,HVACM,GOAL
                * A has entry (or $FF if it was illegal). After return N-bit
                * of CCR indicates whether entry was OK or not.
                * STA ENTRY was used to make N bit reflect sign of ENRTY
                * rather than the result of a compare.
```

```
                   *******************************************************
                   * A2D - Check temp. sensors (via SPI and MC145041)      *
                   *      If TIC = 0, send addr 0 ignore return data       *
                   *      If TIC = 1, send addr 1 return data is ch.0 val *
                   *      If TIC = 2, send addr 2 return data is ch.1 val *
                   *      If TIC > 2, skip A2D routine                     *
                   *  To compensate for sensor & op-amp offset, A/D result *
                   *   will be modified by subtracting an offset constant  *
                   *******************************************************
0309               A2D      EQU   *         Check temp. sensors
0309 b6 a2                  LDA   TIC       If Tic = 0, 1, or 2 write to SPI
030b a1 02                  CMP   #2
030d 22 24                  BHI   XA2D      If Tic > 2; Exit
030f 48                     ASLA            Move TIC # 0-2 to upper nibble
0310 48                     ASLA
0311 48                     ASLA
0312 48                     ASLA            4 bit left shift
0313 3d 0b                  TST   SPSR      Reads SPIF (part of SPIF clear)
0315 17 02                  BCLR  3,PORTC   Drive low true SA/D CE* to 0
0317 b7 0c                  STA   SPDR      Initiates a transfer
                   * Requests conversion of next channel and returns data
                   *   from previous channel Ch.0=Indoor Ch.1=Outdoor
0319 0f 0b fd SPIFLP   BRCLR 7,SPSR,SPIFLP  Wait for SPI Xfer complete
031c 16 02                  BSET  3,PORTC   Drive low true SA/D CE* to 1
031e b6 a2                  LDA   TIC       If 0-Exit, 1 or 2 Read A/D data
0320 27 11                  BEQ   XA2D      0 so exit
0322 b6 0c                  LDA   SPDR      Get A/D data
0324 02 a2 07               BRSET 1,TIC,ADCH1  If Tic=2, data is Ch.1
0327 c0 06 ea               SUB   OFF0      A/D Ch.0; subtract offset
032a b7 ac                  STA   INTMP     Update indoor temperature
032c 20 05                  BRA   XA2D      & Exit
032e c0 06 eb ADCH1         SUB   OFF1      A/D Ch.1; subtract offset
0331 b7 ad                  STA   OUTMP     Update outdoor temperature
0333 81            XA2D     RTS   ** RETURN from A2D **
```

```
               *******************************************************
               * HVAC - Update Fan, Heat, and Cool outputs           *
               *   Low-true outputs will be buffered to drive 24VAC  *
               *    relay coils in HVAC equipment.(high true in final) *
               *   Heat and Cool requests should not permit short-   *
               *    cycle (ie a min delay is required between changes) *
               *    Once Heat or Cool requested, do not turn off for *
               *    at least 30 sec.  Also enforce 30 sec. minimum   *
               *    off time to restart.                             *
               *   Allow ±1° around target temp as hysteresis.       *
               * HVACM = 0 - Off, 1 - Heat, 2 - Cool, 3 - Fan Only   *
               *******************************************************
0334           HVAC    EQU   *         Update Fan, Heat, and Cool outputs
0334 b6 a3             LDA   SEC       Exit unless sec = 0 or 30
0336 27 04             BEQ   DOHVAC    0 so do HVAC
0338 a1 1e             CMP   #30
033a 26 60             BNE   XHVAC     Exit if not 0 or 30
033c b6 aa     DOHVAC  LDA   HVACM     0-off, 1-heat, 2-cool, 3-fan
033e 26 08             BNE   HM1Q      If not 0 go see if 1
0340 b6 02             LDA   PORTC     Fan*,Heat*,Cool*,Beep;ADen*,E,RS,R/W
0342 aa e0             ORA   #$E0      Set fan, heat, cool all high (off)
0344 b7 02             STA   PORTC     Update port
0346 20 54             BRA   XHVAC     & Exit
0348 a1 01     HM1Q    CMP   #1        Check for mode 1 - heat
034a 26 23             BNE   HM2Q      If not go see if 2
034c 1a 02             BSET  5,PORTC   Turn off cool output
034e b6 ab             LDA   GOAL      Get target temp
0350 0c 02 0d          BRSET 6,PORTC,HONQ  If not; see if it should be
               * Heat on; turn off when indoor temp > goal + 1
0353 4c                INCA            Goal+1 for hysteresis
0354 b1 ac             CMP   INTMP     GOAL+1 < INTMP ? Turn off ?
0356 24 44             BHS   XHVAC     NO; just leave
0358 1c 02             BSET  6,PORTC   Turn off heat
035a 1e 02             BSET  7,PORTC   Turn off fan
035c 3f b2             CLR   HVACON    Turn off flag to indicate off
035e 20 3c             BRA   XHVAC     Then leave
               * Heat off; turn on when indoor temp < goal - 1
0360 4a        HONQ    DECA            Goal-1 for hysteresis
0361 b1 ac             CMP   INTMP     GOAL-1 > INTMP ? Turn on ?
0363 23 37             BLS   XHVAC     NO; just leave
0365 1f 02             BCLR  7,PORTC   Turn on fan
0367 1d 02             BCLR  6,PORTC   Turn on heat
0369 a6 01             LDA   #1
036b b7 b2             STA   HVACON    Set flag to indicate on
036d 20 2d             BRA   XHVAC     Then leave
036f a1 02     HM2Q    CMP   #2        Check for mode 2 - cool
0371 27 08             BEQ   HCOOL     Branch if cool mode 2
0373 1f 02             BCLR  7,PORTC   Turn on fan
0375 1c 02             BSET  6,PORTC   Turn off heat
0377 1a 02             BSET  5,PORTC   Turn off cool
0379 20 21             BRA   XHVAC     Then leave
037b 1c 02     HCOOL   BSET  6,PORTC   Turn off heat output
037d b6 ab             LDA   GOAL      Get target temp
037f 0a 02 0d          BRSET 5,PORTC,CONQ  If not; see if it should be
```

```
              * Cool on; turn off when indoor temp < goal - 1
0382 4c                INCA            Goal-1 for hysteresis
0383 b1 ac             CMP   INTMP    GOAL-1 > INTMP ? Turn off ?
0385 23 15             BLS   XHVAC    NO; just leave
0387 1a 02             BSET  5,PORTC  Turn off cool
0389 1e 02             BSET  7,PORTC  Turn off fan
038b 3f b2             CLR   HVACON   Turn off flag to indicate off
038d 20 0d             BRA   XHVAC    Then leave
              * Cool off; turn on when indoor temp > goal + 1
038f 4a       CONQ     DECA            Goal+1 for hysteresis
0390 b1 ac             CMP   INTMP    GOAL+1 < INTMP ? Turn on ?
0392 24 08             BHS   XHVAC    NO; just leave
0394 1f 02             BCLR  7,PORTC  Turn on fan
0396 1b 02             BCLR  5,PORTC  Turn on cool
0398 a6 01             LDA   #1
039a b7 b2             STA   HVACON   Set flag to indicate on
039c 81       XHVAC    RTS   ** RETURN from HVAC **

              *********************************************************
              * LCD - LCD Display Update                              *
              *  If value is being set now, display ENTRY rather than *
              *  the current value and flash it like time colon.      *
              *  Flash time colon if time not being set now (else:on) *
              *  Update current time if time not being set now        *
              *  Update HVAC active '*' unless HVAC mode being set now*
              *  Flash value to set if user is changing a setting     *
              *********************************************************
039d          LCD      EQU   *        LCD Display Update
039d a6 80             LDA   #$80     Left end of 1st row
039f cd 06 20          JSR   WCTRL    Position entry point
03a2 b6 a2             LDA   TIC      50mS periods 0-19
03a4 27 09             BEQ   TIC0     Only update once/sec
03a6 a1 0a             CMP   #10      TIC = 10 at mid second
03a8 26 08             BNE   XLCD     If not 0 or 10, just leave
03aa cd 03 b3          JSR   BLINKR   Blanks colon or value being set
03ad 20 03             BRA   XLCD     Exit
03af cd 04 0f TIC0     JSR   DSPLAY   Update the LCD display
03b2 81       XLCD     RTS            ** RETURN from LCD **
```

```
                 ***
                 * Following subroutines support the LCD main task
                 ***

03b3             BLINKR  EQU   *        Blink colon or user entry
03b3 be b1               LDX   MODE     Mode 0 ?
03b5 26 07               BNE   CIF1     If not see if mode 1
03b7 a6 82               LDA   #$82     Cursor position of colon
03b9 cd 06 20            JSR   WCTRL    Send cursor position to LCD
03bc 20 4b               BRA   SP1      Send 1 ASCII space and leave
03be 5a          CIF1    DECX           Mode 1 ?
03bf 26 07               BNE   CIF2     If not see if mode 2
03c1 a6 80               LDA   #$80     Cursor position of HR
03c3 cd 06 20            JSR   WCTRL    Send cursor position to LCD
03c6 20 3c               BRA   SP2      Send 2 ASCII spaces and leave
03c8 5a          CIF2    DECX           Mode 2 ?
03c9 26 07               BNE   CIF3     If not see if mode 3
03cb a6 83               LDA   #$83     Cursor position of MIN
03cd cd 06 20            JSR   WCTRL    Send cursor position to LCD
03d0 20 32               BRA   SP2      Send 2 ASCII spaces and leave
03d2 5a          CIF3    DECX           Mode 3 ?
03d3 26 07               BNE   CIF4     If not see if mode 4
03d5 a6 86               LDA   #$86     Cursor position of AMPM
03d7 cd 06 20            JSR   WCTRL    Send cursor position to LCD
03da 20 2d               BRA   SP1      Send 1 ASCII space and leave
03dc 5a          CIF4    DECX           Mode 4 ?
03dd 26 07               BNE   CIF5     If not see if mode 5
03df a6 88               LDA   #$88     Cursor position of DAY
03e1 cd 06 20            JSR   WCTRL    Send cursor position to LCD
03e4 20 16               BRA   SP4      Send 4 ASCII spaces and leave
03e6 5a          CIF5    DECX           Mode 5 ?
03e7 26 07               BNE   MUSTB6   If not, mode must be 6
03e9 a6 c0               LDA   #$C0     Cursor position of HVAC Mode
03eb cd 06 20            JSR   WCTRL    Send cursor position to LCD
03ee 20 07               BRA   SP5      Send 5 ASCII spaces and leave
03f0 a6 c6       MUSTB6  LDA   #$C6     Must be mode 6
03f2 cd 06 20            JSR   WCTRL    Cursor position of Goal Temp
03f5 20 0d               BRA   SP2      Send 2 ASCII spaces and leave
03f7 a6 20       SP5     LDA   #$20     ASCII space <sp>
03f9 cd 06 3a            JSR   WDAT     Send a space to LCD
03fc a6 20       SP4     LDA   #$20     ASCII space <sp>
03fe cd 06 3a            JSR   WDAT     Send a space to LCD
0401 cd 06 3a            JSR   WDAT     Send a space to LCD
0404 a6 20       SP2     LDA   #$20     ASCII space <sp>
0406 cd 06 3a            JSR   WDAT     Send a space to LCD
0409 a6 20       SP1     LDA   #$20     ASCII space <sp>
040b cd 06 3a            JSR   WDAT     Send a space to LCD
040e 81                  RTS            ** RETURN from BLINKR **
```

```
                   ***********************************************************
                   * DSPLAY - Writes full 40 character display of current   *
                   *       system conditions to the LCD display peripheral *
                   * Following is a typical LCD display...                 *
                   * 1 2 : 0 0   A   S U N    I N   7 5 ° F *
                   *   O F F    7 2 °    O U T   1 0 2 ° F *
                   *****************************************
040f a6 00     DSPLAY  LDA    #$00     Left end of 1st line on LCD
0411 cd 06 20          JSR    WCTRL    Position entry point
0414 be b1             LDX    MODE     Use for mode compares
0416 b6 a6             LDA    HR
0418 a3 01             CPX    #1       Mode= HR set ?
041a 26 02             BNE    AE1      Skip if not 1
041c b6 a5             LDA    ENTRY    Use ENTRY rather than HR
041e cd 06 a6 AE1      JSR    CNVERT   Convert HRs to ASCII
0421 cd 06 56          JSR    SHOW2    Display as 2 digits
0424 a6 3a             LDA    #':      ASCII colon
0426 cd 06 3a          JSR    WDAT     To LCD
0429 b6 a7             LDA    MIN
042b a3 02             CPX    #2       Mode= MIN set ?
042d 26 02             BNE    AE2      Skip if not 2
042f b6 a5             LDA    ENTRY    Use ENTRY rather than MIN
0431 cd 06 a6 AE2      JSR    CNVERT   Convert MINs to ASCII
0434 cd 06 56          JSR    SHOW2    Display as 2 digits
0437 a6 20             LDA    #$20     ASCII <sp>
0439 cd 06 3a          JSR    WDAT     <sp> to LCD
043c b6 a8             LDA    AMPM     Current AMPM indicator
043e a3 03             CPX    #3       Mode= AMPM set ?
0440 26 02             BNE    AE3      Skip if not 3
0442 b6 a5             LDA    ENTRY    Use ENTRY rather than AMPM
0444 4d        AE3     TSTA            Check for AM (0)
0445 26 04             BNE    ITSPM    If not its PM
0447 a6 41             LDA    #'A      ASCII A
0449 20 02             BRA    SHOWAP   Display A for AM
044b a6 50     ITSPM   LDA    #'P      If it wasn't AM
044d cd 06 3a SHOWAP   JSR    WDAT     Show A or P
0450 a6 20             LDA    #$20     ASCII <sp>
0452 cd 06 3a          JSR    WDAT     To LCD
0455 a6 fc             LDA    #-4      Offset from MDAY
0457 a3 04             CPX    #4       Mode= DAY set ?
0459 26 04             BNE    AE4      Skip if not 4
045b be a5             LDX    ENTRY    Use ENTRY rather than DAY
045d 20 02             BRA    DAYLP    Print Entry day
045f be a9     AE4     LDX    DAY      DAY = 1 to 7
0461 ab 04     DAYLP   ADD    #4       Advance pointer to next MDAY entry
0463 5a                DECX            1->0 or n->(n-1)
0464 26 fb             BNE    DAYLP    Loop till X=0 (A will = 4*DAY)
0466 97                TAX             Move offset to X
0467 d6 06 8a SHODAY   LDA    MDAY,X   Get next char
046a a1 04             CMP    #4       End of message ?
046c 27 06             BEQ    DUNDAY   If done printing day
046e cd 06 3a          JSR    WDAT     Send char to LCD
0471 5c                INCX            Point at next char
0472 20 f3             BRA    SHODAY   Loop till $04 found
0474 5f        DUNDAY  CLRX            Loop index
```

```
0475 d6 06 80 LPSIN   LDA   MSINS,X  Get next ASCII char
0478 cd 06 3a         JSR   WDAT     Loop prints '  IN '
047b 5c               INCX
047c a3 05            CPX   #5
047e 26 f5            BNE   LPSIN    Loop till 5 chars
0480 b6 ac            LDA   INTMP    Indoor temp
0482 cd 06 a6         JSR   CNVERT   Convert to ASCII
0485 cd 06 56         JSR   SHOW2    Display as 2 digits
0488 cd 06 5f         JSR   LCDDF    Display '°F'
048b a6 c0            LDA   #$C0     Left end of 2nd line
048d cd 06 20         JSR   WCTRL    Reposition entry point
0490 a6 20            LDA   #$20     ASCII <sp>
0492 3d b2            TST   HVACON   Heat/cool running ?
0494 27 02            BEQ   ARNAST   If not go around asterisk
0496 a6 2a            LDA   #'*      ASCII asterisk
0498 cd 06 3a ARNAST  JSR   WDAT     Show <sp> or *
049b 5f               CLRX           Message offset from MHVAC
049c b6 b1            LDA   MODE     Get Mode in A
049e a1 05            CMP   #5       Mode= HVACM set ?
04a0 26 04            BNE   AE5      Skip if not 5
04a2 b6 a5            LDA   ENTRY    Use ENTRY rather than HVACM
04a4 20 02            BRA   AE5B
04a6 b6 aa    AE5     LDA   HVACM    HVAC mode
04a8 27 0e    AE5B    BEQ   HVD      If HVACM=0 display 'OFF  '
04aa ae 06            LDX   #6       Offset to 'HEAT '
04ac a1 01            CMP   #1       Heat mode ?
04ae 27 08            BEQ   HVD      If so; display
04b0 ae 0c            LDX   #12      Offset to 'COOL '
04b2 a1 02            CMP   #2       Cool mode ?
04b4 27 02            BEQ   HVD      If so; display
04b6 ae 12            LDX   #18      Offset to 'FAN  ' (must be)
04b8 d6 06 68 HVD     LDA   MHVAC,X
04bb a1 04            CMP   #4       End of message ?
04bd 27 06            BEQ   DUNHVD   If so, skip ahead
04bf cd 06 3a         JSR   WDAT     Else display nxt char
04c2 5c               INCX           Point at next
04c3 20 f3            BRA   HVD      Continue loop
04c5 b6 ab    DUNHVD  LDA   GOAL     Goal temp setting
04c7 be b1            LDX   MODE     Get mode in X
04c9 a3 06            CPX   #6       Mode= GOAL set ?
04cb 26 02            BNE   AE6      Skip if not 6
04cd b6 a5            LDA   ENTRY    Use ENTRY rather than GOAL
04cf cd 06 a6 AE6     JSR   CNVERT   Convert to ASCII
04d2 cd 06 56         JSR   SHOW2    Display as 2 digits
04d5 cd 06 5f         JSR   LCDDF    Display '°F'
04d8 5f               CLRX           Loop index
04d9 d6 06 85 LPSOT   LDA   MSOUT,X  Get message character
04dc cd 06 3a         JSR   WDAT     Send to LCD
04df 5c               INCX           Nxt char of ' OUT '
04e0 a3 05            CPX   #5       Check for done
04e2 26 f5            BNE   LPSOT    Loop for 5 characters
04e4 b6 ad            LDA   OUTMP    Outdoor temp
04e6 cd 06 a6         JSR   CNVERT   Convert to ASCII
04e9 cd 06 52         JSR   SHOW3    Display as 3 digits
04ec cd 06 5f         JSR   LCDDF    Display '°F'
04ef 81               RTS            ** RETURN from DSPLAY **
```

```
0600                ORG   $0600    Temp ORG to get subs away from main

              ********************************
              *                              *
              * SUBROUTINES & CONSTANT TABLES *
              *                              *
              ********************************

              * Keypad Correspondance Table
              *  1st entry of each pair is Row/Col bit pattern
              *  2nd entry of each pair is ASCII equiv of key
              *
              * COL # ->   1  2  3  4
              *            v  v  v  v
              *                         This is layout of keypad
              * ROW 1 ->   1  2  3  A
              * ROW 2 ->   4  5  6  B
              * ROW 3 ->   7  8  9  C
              * ROW 4 ->   <  0  >  !
              *
              * Port B layout is...
              *  R4,R3,R2,R1; C1,C2,C3,C4  R's=ins, C's=outs
              *
0600 18 31    KYTBL   FCB   $18,'1   Row 1, Col 1 (Top Left)
0602 28 34            FCB   $28,'4   Row 2, Col 1
0604 48 37            FCB   $48,'7   Row 3, Col 1
0606 88 3c            FCB   $88,'<   Row 4, Col 1
0608 14 32            FCB   $14,'2   Row 1, Col 2
060a 24 35            FCB   $24,'5   Row 2, Col 2
060c 44 38            FCB   $44,'8   Row 3, Col 2
060e 84 30            FCB   $84,'0   Row 4, Col 2
0610 12 33            FCB   $12,'3   Row 1, Col 3
0612 22 36            FCB   $22,'6   Row 2, Col 3
0614 42 39            FCB   $42,'9   Row 3, Col 3
0616 82 3e            FCB   $82,'>   Row 4, Col 3
0618 11 41            FCB   $11,'A   Row 1, Col 4
061a 21 42            FCB   $21,'B   Row 2, Col 4
061c 41 43            FCB   $41,'C   Row 3, Col 4
061e 81 21            FCB   $81,'!   Row 4, Col 4 (Bot Right)
```

```
              ********************************************************
              * WCTRL - Write control word to LCD peripheral         *
              *       Enter with control word in accumulator         *
              *       Return with original value of X                *
              *       Delay ~4.5mS if A=$01 or $02 else delay ~120µS *
              ********************************************************
0620 bf a1    WCTRL    STX   TEMPX    Save X
0622 b7 00             STA   PORTA    Write control word to LCD
0624 14 02             BSET  2,PORTC  E->1
0626 15 02             BCLR  2,PORTC  E->0
0628 ae 14             LDX   #20      20*6~*1µS/~=120µS
062a 5a       L120U    DECX           Delay loop ~120µS
062b 26 fd             BNE   L120U    20-19,19-18...1-0
062d a1 02             CMP   #$02     Commands $01 & $02 req extra delay
062f 22 06             BHI   ARN5M    If command > $02 skip long delay
0631 cd 06 39 L5M      JSR   ANRTS    JSR+RTS TAKES 12~ (just want delay)
0634 5a                DECX           TAKES 3~ (X=0->1 on first pass)
0635 26 fa             BNE   L5M      3~ Loop 256*18~*1µS/~=4.608mS Delay
0637 be a1    ARN5M    LDX   TEMPX    Restore X
0639 81       ANRTS    RTS            ** RETURN **

              ********************************************************
              * WDAT - Write data word to LCD peripheral             *
              *       Enter with data word in accumulator            *
              *       Return with original values of X & A           *
              *       Delay ~120µS after data write                  *
              ********************************************************
063a bf a1    WDAT     STX   TEMPX    Save X
063c b7 a0             STA   TEMPA    Save A
063e b7 00             STA   PORTA    Write data word to LCD
0640 12 02             BSET  1,PORTC  RS->1
0642 14 02             BSET  2,PORTC  E->1
0644 15 02             BCLR  2,PORTC  E->0
0646 13 02             BCLR  1,PORTC  RS->0
0648 ae 14             LDX   #20      20*6~*1µS/~=120µS
064a 5a       L120     DECX           Delay loop ~120µS
064b 26 fd             BNE   L120     20-19,19-18...1-0
064d b6 a0             LDA   TEMPA    Restore A
064f be a1             LDX   TEMPX    Restore X
0651 81                RTS            ** RETURN **
```

```
                  ***************************************************
                  * SHOW3 - Display 3 ASCII chars on LCD            *
                  *       ASC100, ASC10; ASC1                       *
                  * SHOW2 - Display 2 ASCII chars on LCD            *
                  *       ASC10; ASC1                               *
                  ***************************************************
0652 b6 ae        SHOW3    LDA   ASC100    Get ASCII 100's digit
0654 ad e4                 BSR   WDAT      Send to LCD
0656 b6 af        SHOW2    LDA   ASC10     Get ASCII 10's digit
0658 ad e0                 BSR   WDAT      Send to LCD
065a b6 b0                 LDA   ASC1      Get ASCII 1's digit
065c ad dc                 BSR   WDAT      Send to LCD
065e 81                    RTS             ** RETURN **

                  ***************************************************
                  * LCDDF - Display °F on LCD                       *
                  ***************************************************
065f a6 df        LCDDF    LDA   #$DF      Get ASCII degrees symbol
0661 ad d7                 BSR   WDAT      Send to LCD
0663 a6 46                 LDA   #'F       Get ASCII capitol F
0665 ad d3                 BSR   WDAT      Send to LCD
0667 81                    RTS             ** RETURN **

                  * Normal LCD display format...
                  * H H : M M   A   D A Y   I N   1 0 0 ° F
                  * _ H E A T _ 7 2 ° _ _ O U T   - 2 2 ° F
                  *       1st line of display is $00 (left) - $13
                  *       2nd line of display is $40 - $53

                  * Miscellaneous LCD message segments (Used in DSPLAY sub)
0668 4f 46 46 20 20 MHVAC  FCC   'OFF  '   These 4 messages accessed by
066d 04                    FCB   $04       X offset from MHVAC.  $04 is
066e 48 45 41 54 20        FCC   'HEAT '   used to mark the end of a string
0673 04                    FCB   $04
0674 43 4f 4f 4c 20        FCC   'COOL '
0679 04                    FCB   $04
067a 46 41 4e 20 20        FCC   'FAN  '
067f 04                    FCB   $04
0680 20 20 49 4e 20  MSINS FCC   '  IN '
0685 20 4f 55 54 20  MSOUT FCC   ' OUT '
068a 53 55 4e         MDAY FCC   'SUN'     These messages accessed by
068d 04                    FCB   $04       X offset from MDAY.  $04 is
068e 4d 4f 4e              FCC   'MON'     used to mark the end of a string
0691 04                    FCB   $04
0692 54 55 45              FCC   'TUE'
0695 04                    FCB   $04
0696 57 45 44              FCC   'WED'
0699 04                    FCB   $04
069a 54 48 55              FCC   'THU'
069d 04                    FCB   $04
069e 46 52 49              FCC   'FRI'
06a1 04                    FCB   $04
06a2 53 41 54              FCC   'SAT'
06a5 04                    FCB   $04
```

```
                *********************************************************
                * CNVERT - Convert a binary value to ASCII              *
                *       Enter with binary value in A                    *
                *       Result stored in ASC100, ASC10, ASC1            *
                *       ASC100 (100's digit) defaults to blank (<sp>)   *
                *        but could be 1 or minus (-) depending on valu *
                *       ASC10 and ASC1 digits default to zeros          *
                *       Result can be -99 through 127.                  *
                *********************************************************
06a6 b7 a0      CNVERT  STA   TEMPA     Save original binary value
06a8 a6 20              LDA   #$20      ASCII <sp>
06aa b7 ae              STA   ASC100    Tenative 100's digit
06ac a6 30              LDA   #'0       ASCII zero
06ae b7 af              STA   ASC10     Tenative 10's
06b0 b7 b0              STA   ASC1      Tenative 1's
06b2 b6 a0              LDA   TEMPA     Get value to convert
06b4 2a 19              BPL   CVPOS     Branch if value positive
06b6 a6 2d              LDA   #'-       ASCII minus sign
06b8 b7 ae              STA   ASC100
06ba b6 a0              LDA   TEMPA     Get orig value again
06bc 3c af      LP10S   INC   ASC10     Loop to find 10's digit
06be ab 0a              ADD   #10       Trial addition
06c0 2b fa              BMI   LP10S     Loop till addition fails
06c2 27 25              BEQ   XVERT     If 0 conversion done; exit
06c4 3a af              DEC   ASC10     Too far; back up
06c6 a0 0a              SUB   #10       Now between -9 & -1
06c8 40                 NEGA            Change to positive
06c9 bb b0              ADD   ASC1      Add to 1's digit
06cb b7 b0              STA   ASC1      Update RAM location
06cd 20 1a              BRA   XVERT     Conversion done; exit

06cf a1 64      CVPOS   CMP   #100      Value >100 ?
06d1 25 08              BLO   LPAS10    If less; skip 100's
06d3 a6 31              LDA   #'1
06d5 b7 ae              STA   ASC100    Put ASCII 1 in 100's
06d7 b6 a0              LDA   TEMPA     Get value again
06d9 a0 64              SUB   #100      Take 100 away
06db 3c af      LPAS10  INC   ASC10     Increments 10's
06dd a0 0a              SUB   #10       Trial subtraction
06df 2a fa              BPL   LPAS10    Loop till trial sub fails
06e1 3a af              DEC   ASC10     Too far
06e3 ab 0a              ADD   #10       Add back, now 0-9
06e5 bb b0              ADD   ASC1      Add to ASCII 1's
06e7 b7 b0              STA   ASC1      Update RAM location
06e9 81         XVERT   RTS             ** RETURN from CNVERT **


                * A/D Offsets to compensate sensors
                *       Analog temp = (A/D reading) - (Offset)
06ea 3c         OFF0    FCB   60        Offset correction for sensor 1
06eb 3c         OFF1    FCB   60        Offset correction for sensor 2
                ***********

1ffe                    ORG   $1FFE     Reset vector address
1ffe 01 00              FDB   INIT      Reset vector
```

4

APPENDIX A
INSTRUCTION SET DETAILS

This appendix contains complete detailed information for all M68HC05 instructions. The instructions are arranged in alphabetical order with the instruction mnemonic set in larger type for easy reference.

The nomenclature listed below is used in the following definitions:

(a) Operators

( ) = Contents of Register or Memory Location Shown inside Parentheses
← = Is Loaded with (read:"gets")
↑ = Is Pulled from Stack
↓ = Is Pushed onto Stack
• = Boolean AND
+ = Arithmetic Addition (Except Where Used as Inclusive-OR in Boolean Formula)
⊕ = Boolean Exclusive-OR
× = Multiply
: = Concatenate
− = Negate (Twos Complement)

(b) Registers in the MPU

ACCA = Accumulator
CCR = Condition Code Register
X = Index Register
PC = Program Counter
PCH = Program Counter, Higher Order (Most Significant) 8 Bits
PCL = Program Counter, Lower Order (Least Significant) 8 Bits
SP = Stack Pointer

(c) Memory and Addressing

M = A memory location or absolute data, depending on addressing mode
Rel = Relative offset (i.e., the twos-complement number stored in the last byte of machine code corresponding to a branch instruction)

(d) Bits in the Condition Code Register (CCR)

H = Half Carry, Bit 4
I = Interrupt Mask, Bit 3
N = Negative Indicator, Bit 2
C = Carry/Borrow, Bit 1
Z = Zero Indicator, Bit 0

(e) Status of individual bits BEFORE execution of an instruction (n = 7, 6, 5, . . . 0)

An = Bit n of ACCA
Xn = Bit n of X
Mn = Bit n of M

(f) Status of individual bits AFTER the execution of an instruction

Rn = Bit n of the result (n = 7, 6, 5, . . . 0)

(g) Notation used in CCR activity summary figures

— = Bit not affected
0 = Bit forced to zero
1 = Bit forced to one
↕ = Bit set or cleared according to results of operation

(h) Notation used in machine coding

dd = Low-order 8-bits of a direct address $0000–$00FF (high byte assumed to be $0000)
ee = Upper 8 bits of 16-bit offset
ff = Lower 8 bits of 16-bit offset or 8-bit offset
ii = One byte of immediate data
hh = High-order byte of 16-bit extended address
ll = Low-order byte of 16-bit extended address
rr = Relative offset

(i) Notation used in source forms

(opr) = Operand (one or two bytes depending on address mode)
(rel) = Relative offset used in branch and bit manipulation instructions

ADC Add with Carry ADC

Operation: ACCA ← (ACCA) + (M) + (C)

Description: Adds the contents of the C bit to the sum of the contents of ACCA and M and places the result in ACCA.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	↕	—	↕	↕	↕

H $A3 \cdot M3 + M3 \cdot \overline{R3} + \overline{R3} \cdot A3$
Set if there was a carry from bit 3; cleared otherwise.

N $R7$
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if all bits of the result are cleared; cleared otherwise.

C $A7 \cdot M7 + M7 \cdot \overline{R7} + \overline{R7} \cdot A7$
Set if there was a carry from the MSB of the result; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
ADC (opr)	IMM	A9	ii	2
ADC (opr)	DIR	B9	dd	3
ADC (opr)	EXT	C9	hh ll	4
ADC,X	IX	F9		3
ADC (opr),X	IX1	E9	ff	4
ADC (opr),X	IX2	D9	ee ff	5

ADD Add without Carry ADD

Operation: ACCA ← (ACCA) + (M)

Description: Adds the contents of M to the contents of ACCA and places the result in ACCA.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	↕	—	↕	↕	↕

H $A3 \cdot M3 + M3 \cdot \overline{R3} + \overline{R3} \cdot A3$
Set if there was a carry from bit 3; cleared otherwise.

N $R7$
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if all bits of the result are cleared; cleared otherwise.

C $A7 \cdot M7 + M7 \cdot \overline{R7} + \overline{R7} \cdot A7$
Set if there was a carry from the MSB of the result; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
ADD (opr)	IMM	AB	ii	2
ADD (opr)	DIR	BB	dd	2
ADD (opr)	EXT	CB	hh ll	3
ADD,X	IX	FB		1
ADD (opr),X	IX1	EB	ff	2
ADD (opr),X	IX2	DB	ee ff	3

AND Logical AND AND

Operation: ACCA $\leftarrow$ (ACCA) • (M)

Description: Performs the logical AND between the contents of ACCA and the contents of M and places the result in ACCA. (Each bit of ACCA after the operation will be the logical AND of the corresponding bits of M and of ACCA before the operation.)

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

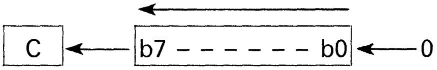
Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if all bits of the result are cleared; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
AND (opr)	IMM	A4	ii	2
AND (opr)	DIR	B4	dd	3
AND (opr)	EXT	C4	hh ll	4
AND,X	IX	F4		3
AND (opr),X	IX1	E4	ff	4
AND (opr),X	IX2	D4	ee ff	5

ASL Arithmetic Shift Left (Same as LSL) ASL

Operation:



Description: Shifts all bits of the ACCA, X, or M one place to the left. Bit 0 is loaded with a zero. The C bit in the CCR is loaded from the most significant bit of ACCA, X, or M.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N R7
Set if MSB of result is set; cleared otherwise.

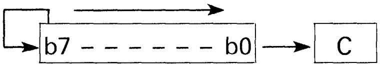
Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if all bits of the result are cleared; cleared otherwise.

C b7
Set if, before the shift, the MSB of ACCA, X, or M was set; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
ASLA	INH (A)	48		3
ASLX	INH (X)	58		3
ASL (opr)	DIR	38	dd	5
ASL, X	IX	78		5
ASL (opr),X	IX1	68	ff	6

Operation:



Description: Shifts all of ACCA, X, or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit of the CCR. This operation effectively divides a twos-complement value by two without changing its sign. The carry bit can be used to round the result.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if all bits of the result and cleared; cleared otherwise.

C b0
Set if, before the shift, the LSB of ACCA, X, or M was set; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
ASRA	INH (A)	47		3
ASRX	INH (X)	57		3
ASR (opr)	DIR	37	dd	5
ASR, X	IX	77		5
ASR (opr),X	IX1	67	ff	6

BCC Branch if Carry Clear (Same as BHS) BCC

Operation: PC $\leftarrow$ (PC)+\$0002+Rel if (C)=0

Description: Tests the state of the C bit in the CCR and causes a branch if C is clear.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BCC (rel)	REL	24	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r=register (ACCA or X)
m=memory operand

BCLR n   Clear Bit(s) in Memory   BCLR n

Operation: Mn ← 0

Description: Clear bit n (n = 7, 6, 5, . . . 0) in location M. All other bits in M are unaffected. M can be any RAM or I/O register address in the $0000 to $00FF area of memory (i.e., direct addressing mode is used to specify the address of the operand).

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code		HCMOS Cycles
		Opcode	Operand(s)	
BCLR 0,(opr)	DIR (bit 0)	11	dd	5
BCLR 1,(opr)	DIR (bit 1)	13	dd	5
BCLR 2,(opr)	DIR (bit 2)	15	dd	5
BCLR 3,(opr)	DIR (bit 3)	17	dd	5
BCLR 4,(opr)	DIR (bit 4)	19	dd	5
BCLR 5,(opr)	DIR (bit 5)	1B	dd	5
BCLR 6,(opr)	DIR (bit 6)	1D	dd	5
BCLR 7,(opr)	DIR (bit 7)	1F	dd	5

Branch if Carry Set
(Same as BLO)

Operation: PC ← (PC) + \$0002 + Rel if (C) = 1

Description: Tests the state of the C bit in the CCR and causes a branch if C is set.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
BCS (rel)	REL	25	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary Branch			Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BEQ Branch if Equal BEQ

Operation: PC ← (PC) + $0002 + Rel if (Z) = 1

Description: Tests the state of the Z bit in the CCR and causes a branch if Z is set. Following a compare or subtract instruction, BEQ will cause a branch if the arguments were equal.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BEQ (rel)	REL	27	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

Branch if Half Carry Clear

Operation: PC ← (PC) + $0002 + Rel if (H) = 0

Description: Tests the state of the H bit and causes a branch if H is clear. This instruction is used in algorithms involving BCD numbers.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
BHCC (rel)	REL	28	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BHCS Branch if Half Carry Set BHCS

Operation: PC ← (PC) + $0002 + Rel if (H) = 1

Description: Tests the state of the H bit and causes a branch if H is clear. This instruction is used in algorithms involving BCD numbers. See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BHCS (rel)	REL	29	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

Branch if Higher

Operation: PC $\leftarrow$ (PC) + \$0002 + Rel if (C) + (Z) = 0
i.e., if (ACCX) > (M) (unsigned binary numbers)

Description: Causes a branch if both C and Z are cleared. If the BHI instruction is executed immediately after execution of a CMP or SUB instruction, the branch will occur if the unsigned binary number represented by ACCA was greater than the unsigned binary number represented by M.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code		HCMOS Cycles
		Opcode	Operand(s)	
BHI (rel)	REL	22	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

Branch if Higher or Same (Same as BCC)

Operation: PC $\leftarrow$ (PC) + \$0002 + Rel if (C) = 0
i.e., if (ACCA) $\geq$ (M) (unsigned binary numbers)

Description: If the BHS instruction is executed immediately after execution of a CMP or SUB instruction, the branch will occur if the unsigned binary number represented by ACCA was greater than or equal to the unsigned binary number represented by M.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BHS (rel)	REL	24	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary Branch			Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BIH Branch if Interrupt Pin is High BIH

Operation: PC ← (PC) + $0002 + Rel if $\overline{IRQ} = 1$

Description: Tests the state of the external interrup pin and causes a branch if the pin is high.

See BRA instsruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BIH (rel)	REL	2F	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{IRQ}$ Pin High	—	BIH	2F	$\overline{IRQ}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

A

Branch if Interrupt Pin is Low

Operation: PC ← (PC) + $0002 + Rel if $\overline{\text{IRQ}}=0$

Description: Tests the state of the external interrupt pin and branchs if it is low.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BIL (rel)	REL	2E	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary Branch			Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BIT   Bit Test   BIT

Operation: (ACCX)•(M)

Description: Performs the logical AND comparison of the contents of ACCA and the contents of M, and modifies the condition codes accordingly. Neither the contents of ACCA or M are altered. (Each bit of the result of the AND would be the logical AND of the corresponding bits of ACCA and M).

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BIT (opr)	IMM	A5	ii	2
BIT (opr)	DIR	B5	dd	3
BIT (opr)	EXT	C5	hh ll	4
BIT,X	IX	F5		3
BIT (opr),X	IX1	E5	ff	4
BIT (opr),X	IX2	D5	ee ff	5

BLO **Branch if Lower (Same as BCS)** BLO

Operation: PC ← (PC) + $0002 + Rel if (C) = 1
i.e., if (ACCX)<(M) (unsigned binary numbers)

Description: If the BLO instruction is executed immediately after execution of a CMP or SUB instruction, the branch will occur if the unsigned binary number represented by ACCA was less than the unsigned binary number represented by M.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BLO (rel)	REL	25	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BLS Branch if Lower or Same BLS

Operation: PC ← (PC) + \$0002 + Rel if [(C) + (Z)] = 1
i.e., if (ACCA)≤(M) (unsigned binary numbers)

Description: Causes a branch if (C is set) or (Z is set). If the BLS instruction is executed immediately after execution of a CMP or SUB instruction, the branch will occur if and only if the unsigned binary number represented by ACCA was less than or equal to the unsigned binary number represented by M.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
BLS (rel)	REL	23	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BMC Branch if Interrupt Mask is Clear BMC

Operation: PC ← (PC) + $0002 + Rel if I = 0

Description: Tests the state of the I bit in the CCR and causes a branch if I is clear (i.e., if interrupts are enabled). See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BMC (rel)	REL	2C	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

Branch if Minus

Operation: PC ← (PC) + $0002 + Rel if (N) = 1

Description: Tests the state of the N bit in the CCR and causes a branch if N is set.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BMI (rel)	REL	2B	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BMS Branch if Interrupt Mask is Set BMS

Operation: PC ← (PC)+$0002+Rel if (I)=1

Description: Tests the state of the I bit in the CCR and causes a branch if I is set (i.e., if interrupts are disabled).

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BMS (rel)	REL	2D	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r=register (ACCA or X)
m=memory operand

BNE Branch if Not Equal to Zero BNE

Operation: PC $\leftarrow$ (PC) + \$0002 + Rel if (Z) = 0

Description: Tests the state of the Z bit in the CCR and causes a branch if Z is clear. Following a compare or subtract instruction, BEQ will cause a branch if the arguments were not equal.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BNE (rel)	REL	26	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BPL　Branch if Plus　BPL

Operation: PC ← (PC) + $0002 + Rel　　if (N) = 0

Description: Tests the state of the N bit in the CCR and causes a branch if N is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
BPL (rel)	REL	2A	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary Branch			Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BRA — Branch Always

Operation: PC ← (PC) + $0002 + Rel

Description: Unconditional branch to the address given by the foregoing formula, in which Rel is the relative offset stored as a twos-complement number in the last byte of machine code corresponding to the branch instruction. PC is the address of the opcode for the branch instruction.

The source program specifies the destination of any branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler calculates the relative address, Rel, from the absolute address and the current value of the location counter.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
BRA (rel)	REL	20	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary Branch			Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BRCLR n Branch if Bit n Clear BRCLR n

Operation: PC ← (PC) + $0003 + Rel if bit n of M = 0

Description: Tests bit n (n = 7, 6, 5, . . . 0) of location M and branches if the bit is clear. M can be any RAM or I/O register address in the $0000 to $00FF area of memory (i.e., direct addressing mode is used to specify the address of the operand).

The C bit is set to the state of the bit tested. When used along with an appropriate rotate instruction, **BRCLR n** provides an easy method for performing serial to parallel conversions.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	↕

C Set if Mn = 1; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
BRCLR 0,(opr)	DIR (bit 0)	01	dd	rr	5
BRCLR 1,(opr)	DIR (bit 1)	03	dd	rr	5
BRCLR 2,(opr)	DIR (bit 2)	05	dd	rr	5
BRCLR 3,(opr)	DIR (bit 3)	07	dd	rr	5
BRCLR 4,(opr)	DIR (bit 4)	09	dd	rr	5
BRCLR 5,(opr)	DIR (bit 5)	0B	dd	rr	5
BRCLR 6,(opr)	DIR (bit 6)	0D	dd	rr	5
BRCLR 7,(opr)	DIR (bit 7)	0F	dd	rr	5

Branch Never

Operation: PC ← (PC) + $0002

Description: Never branches. In effect, this instruction can be considered as a two-byte NOP (no operation) requiring three cycles for execution. Its inclusion in the instruction set is to provide a complement for the BRA instruction. The instruction is useful during program debug to negate the effect of another branch instruction without disturbing the offset byte.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
BRN (rel)	REL	21	rr	3

The following table is a summary of all branch instructions.

Test	Boolean	Mnemonic	Opcode	Complementary		Branch	Comment
r>m	C+Z=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	C=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	Z=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	C+Z=1	BLS	23	r>m	BHI	22	Unsigned
r<m	C=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	C=1	BCS	25	No Carry	BCC	24	Simple
r=0	Z=1	BEQ	27	r≠0	BNE	26	Simple
Negative	N=1	BMI	2B	Plus	BPL	2A	Simple
I Mask	I=1	BMS	2D	I Mask=0	BMC	2C	Simple
Half Carry	H=1	BHCS	29	No Half Carry	BHCC	28	Simple
$\overline{\text{IRQ}}$ Pin High	—	BIH	2F	$\overline{\text{IRQ}}$ Low	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Unconditional

r = register (ACCA or X)
m = memory operand

BRSET n Branch if Bit n Set BRSET n

Operation: PC ← (PC) + $0003 + Rel if bit n of M = 1

Description: Tests bit n (n = 7, 6, 5, . . . 0) of location M and branches if the bit is set. M can be any RAM or I/O register address in the $0000 to $00FF area of memory (i.e., direct addressing mode is used to specify the address of the operand).

The C bit is set to the state of the bit tested. When used along with an appropriate rotate instruction, **BRSET n** provides an easy method for performing serial to parallel conversions.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	↕

C Set if Mn = 1; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code		HCMOS Cycles
		Opcode	Operand(s)	
BRSET 0,(opr),(rel)	DIR (bit 0)	00	dd rr	5
BRSET 1,(opr),(rel)	DIR (bit 1)	02	dd rr	5
BRSET 2,(opr),(rel)	DIR (bit 2)	04	dd rr	5
BRSET 3,(opr),(rel)	DIR (bit 3)	06	dd rr	5
BRSET 4,(opr),(rel)	DIR (bit 4)	08	dd rr	5
BRSET 5,(opr),(rel)	DIR (bit 5)	0A	dd rr	5
BRSET 6,(opr),(rel)	DIR (bit 6)	0C	dd rr	5
BRSET 7,(opr),(rel)	DIR (bit 7)	0E	dd rr	5

BSET n — Set Bit in Memory — BSET n

Operation: $Mn \leftarrow 1$

Description: Set bit n (n = 7, 6, 5, . . . 0) in location M. All other bits in M are unaffected. M can be any RAM or I/O register address in the $0000 to $00FF area of memory (i.e., direct addressing mode is used to specify the address of the operand).

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
BSET 0,(opr)	DIR (bit 0)	10	dd	5
BSET 1,(opr)	DIR (bit 1)	12	dd	5
BSET 2,(opr)	DIR (bit 2)	14	dd	5
BSET 3,(opr)	DIR (bit 3)	16	dd	5
BSET 4,(opr)	DIR (bit 4)	18	dd	5
BSET 5,(opr)	DIR (bit 5)	1A	dd	5
BSET 6,(opr)	DIR (bit 6)	1C	dd	5
BSET 7,(opr)	DIR (bit 7)	1E	dd	5

BSR Branch to Subroutine BSR

Operation:

PC ← (PC) + $0002	Advance PC to return address
↓(PCL)	Push low-order return onto stack
SP ← (SP) − $0001	
↓(PCH)	Push high-order return onto stack
SP ← (SP) − $0001	
PC ← (PC) + Rel	Load PC with start address of requested subroutine

Description: The program counter is incremented by two from the opcode address, (i.e., so it points to the opcode of the next instruction which will be the return address). The least significant byte of the contents of the program counter (low-order return address) is pushed onto the stack. The stack pointer is then decremented by one. The most significant byte of the contents of the program counter (high-order return address) is pushed onto the stack. The stack pointer is then decremented by one. A branch then occurs to the location specified by the branch offset.

See BRA instruction for further details of the execution of the branch.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
BSR (rel)	REL	AD	rr	3

CLC Clear Carry CLC

Operation: C bit ← 0

Description: Clears the C bit in the CCR. CLC may be used to set up the C bit prior to a shift or rotate instruction involving the C bit.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	0

C 0
Cleared

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
CLC	INH	98	2

CLI Clear Interrupt Mask Bit CLI

Operation: I bit ← 0

Description: Clears the interrupt mask bit in the CCR. When the I bit is clear, interrupts are enabled. There is a one E-clock cycle delay in the clearing mechanism for the I bit so that, if interrupts were previously disabled, the next instruction after a CLI will always be executed, even if there was an interrupt pending prior to execution of the CLI instruction.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	0	—	—	—

I 0
Cleared

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
CLI	INH	9A	2

CLR Clear CLR

Operation: ACCA ⧫ $00 **or:** M ⧫ $00 **or:** X ⧫ $00

Description; The contents of ACCA, M, or X are replaced with zeros.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	0	1	—

N 0
Cleared

Z 1
Set

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
CLRA	INH (A)	4F		3
CLRX	INH (X)	5F		3
CLR (opr)	DIR	3F	dd	5
CLR, X	IX	7F		5
CLR (opr),X	IX1	6F	ff	6

CMP Compare Accumulator with Memory CMP

Operation: (ACCA) − (M)

Description: Compares the contents of ACCA to the contents of M and sets the condition codes, which may be used for arithmetic and logical conditional branching. The contents of both ACCA and M are unchanged.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N $R7$
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise.

C $\overline{A7} \cdot M7 + M7 \cdot R7 + R7 \cdot \overline{A7}$
Set if the absolute value of the contents of memory is larger than the absolute value of the accumulator; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
CMP (opr)	IMM	A1	ii		2
CMP (opr)	DIR	B1	dd		3
CMP (opr)	EXT	C1	hh	ll	4
CMP,X	IX	F1			3
CMP (opr),X	IX1	E1	ff		4
CMP (opr),X	IX2	D1	ee	ff	5

Operation: $ACCA \leftarrow (\overline{ACCA}) = \$FF - (ACCA)$ **or:** $M \leftarrow (\overline{M}) = \$FF - (M)$ **or:**
$X \leftarrow \overline{X} = \$FF - (X)$

Description: Replaces the contents of ACCA, X, or M with its ones complement. (Each bit of the contents of ACCA, X, or M is replaced with the complement of that bit.)

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	1

N $R7$
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is \$00; cleared otherwise.

C 1
Set

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
COMA	INH (A)	43		3
COMX	INH (X)	53		3
COM (opr)	DIR	33	dd	5
COM, X	IX	73		5
COM (opr),X	IX1	63	ff	6

CPX Compare Index Register with Memory CPX

Operation: (X) − (M)

Description: Compares the contents of the index register with the contents of memory and sets the condition codes, which may be used for arithmetic and logical branching. The contents of both ACCA and M are unchanged.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is \$00; cleared otherwise.

C $\overline{IX7} \cdot M7 + M7 \cdot R7 + R7 \cdot \overline{IX7}$
Set if the absolute value of the contents of memory is larger than the absolute value of the index register; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
CPX (opr)	IMM	A3	ii		2
CPX (opr)	DIR	B3	dd		3
CPX (opr)	EXT	C3	hh	ll	4
CPX,X	IX	F3			3
CPX (opr),X	IX1	E3	ff		4
CPX (opr),X	IX2	D3	ee	ff	5

DEC Decrement DEC

Operation: ACCA ← (ACCA) − \$01 **or:** M ← (M) − \$01 **or:** X ← (X) − \$01

Description: Subtract one from the contents of ACCA, X, or M.

The N and Z bits in the CCR are set or cleared according to the result of this operation. The C bit is in the CCR is not affected; therefore, the only branch instructions that are useful following a DEC instruction are BEQ, BNE, BPL, and BMI.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is \$00; cleared otherwise

Source Forms, Addressing Modes Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
DECA	INH (A)	4A	3
DECX	INH (X)	5A	3
DEC (opr)	DIR	3A dd	5
DEC, X	IX	7A	5
DEC (opr),X	IX1	6A ff	6

(DEX is recognized by the Assembler as being equivalent to DECX)

EOR Exclusive-OR EOR

Operation: $ACCA \leftarrow (ACCA) \oplus (M)$

Description: Performs the logical exclusive-OR between the contents of ACCA and the contents of M and places the result in ACCA. (Each bit of ACCA after the operation will be the logical exclusive-OR of the corresponding bits of M and ACCA before the operation.)

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
EOR (opr)	IMM	A8	ii	2
EOR (opr)	DIR	B8	dd	3
EOR (opr)	EXT	C8	hh ll	4
EOR,X	IX	F8		3
EOR (opr),X	IX1	E8	ff	4
EOR (opr),X	IX2	D8	ee ff	5

Operation: ACCA ← (ACCA)+$01 **or:** M ← (M)+$01 **or:** X ← (X)+$01

Description: Add one to the contents of ACCA, X, or M.

The N and Z bits in the CCR are set or cleared according to the results of this operation. The C bit in the CCR is not affected; therefore, the only branch instructions that are useful following a INC instruction are BEQ, BNE, BPL, and BMI.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
INCA	INH (A)	4C		3
INCX	INH (X)	5C		3
INC (opr)	DIR	3C	dd	5
INC, X	IX	7C		5
INC (opr),X	IX1	6C	ff	6

(INX is recognized by the Assembler as being equivalent to INCX)

JMP Jump JMP

Operation: PC ← Effective Address

Description: A jump occurs to the instruction stored at the effective address. The effective address is obtained according to the rules for EXTended, DIRect, or INDexed addressing.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
JMP (opr)	DIR	BC	dd		2
JMP (opr)	EXT	CC	hh	ll	3
JMP, X	IX	FC			2
JMP (opr), X	IX1	EC	ff		3
JMP (opr),X	IX2	DC	ee	ff	4

JSR Jump to Subroutine JSR

Operation:

PC ← (PC) + n	n = 1, 2, 3 depending on address mode
↓ (PCL); SP ← SP − $0001	Push low-order return address onto stack
↓ (PCH); SP ← SP − $0001	Push high-order return address onto stack
PC ← Effective Addr	Load PC with start address of requested subroutine

Description: The program counter is incremented by n so that it points to the opcode of the instruction that follows the JSR instruction (n = 1, 2, or 3 depending on the addressing mode). The PC is then pushed onto the stack, eight bits at a time, least significant byte first. Unused bits in the program counter high byte are stored as ones on the stack. The stack pointer points to the next empty location on the stack. A jump occurs to the instruction stored at the effective address. The effective address is obtained according to the rules for EXTended, DIRect, or INDexed addressing.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
JSR (opr)	DIR	BD	dd		5
JSR (opr)	EXT	CD	hh	ll	6
JSR, X	IX	FD			5
JSR (opr), X	IX1	ED	ff		6
JSR (opr),X	IX2	DD	ee	ff	7

LDA Load Accumulator from Memory LDA

Operation: ACCA ← (M)

Description: Loads the contents of memory into the accumulator. The condition codes are set according to the data.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
LDA (opr)	IMM	A6	ii		2
LDA (opr)	DIR	B6	dd		3
LDA (opr)	EXT	C6	hh	ll	4
LDA,X	IX	F6			3
LDA (opr),X	IX1	E6	ff		4
LDA (opr),X	IX2	D6	ee	ff	5

Load Index Register from Memory

Operation; X ← (M)

Description: Loads the contents of the specified memory location into the index register. The condition codes are set according to the data.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is $00; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
LDX (opr)	IMM	AE	ii	2
LDX (opr)	DIR	BE	dd	3
LDX (opr)	EXT	CE	hh ll	4
LDX,X	IX	FE		3
LDX (opr),X	IX1	EE	ff	4
LDX (opr),X	IX2	DE	ee ff	5

Logical Shift Left
(Same as ASL)

Operation: $\boxed{\text{C}} \leftarrow \overset{\longleftarrow}{\boxed{\text{b7 - - - - - - - b0}}} \leftarrow 0$

Description: Shifts all bits of the ACCA, X, or M one place to the left. Bit 0 is loaded with zero. The C bit is loaded from the most significant bit of ACCA, X, or M.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is $00; cleared otherwise.

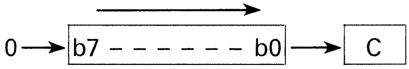
C b7
Set if, before the shift, the MSB of ACCA or M was set; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
LSLA	INH (A)	48		3
LSLX	INH (X)	58		3
LSL (opr)	DIR	38	dd	5
LSL, X	IX	78		5
LSL (opr),X	IX1	68	ff	6

LSR Logical Shift Right LSR

Operation:



Description: Shifts all bits of ACCA, X, or M one place to the right. Bit 7 is loaded with zero. Bit 0 is shifted into the C bit.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	0	⇕	⇕

N 0
Cleared.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise.

C b0
Set if, before the shift, the LSB of ACCA, X, or M was set; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
LSRA	INH (A)	44		3
LSRX	INH (X)	54		3
LSR (opr)	DIR	34	dd	5
LSR, X	IX	74		5
LSR (opr),X	IX1	64	ff	6

MUL Multiply Unsigned MUL

Operation: $X:A \leftarrow X \times A$

Description: Multiplies the eight bits in the index register by the eight bits in the accumulator to obtain a 16-bit unsigned number in the concatenated index register and accumulator. After the operation, X contains the upper 8 bits of the 16-bit result.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	0	—	—	—	0

H 0
Cleared

C 0
Cleared

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
MUL	INH	42		11

NEG Negate NEG

Operation: ACCA ← −(ACCA); **or:** X ← −(X); **or:** M ← −(M)

Description: Replaces the contents of ACCA, X, or M with its twos complement. Note that the value $80 is left unchanged.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N $R7$
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is $00; cleared otherwise.

C $R7+R6+R5+R4+R3+R2+R1+R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. The C bit will be set in all cases **except** when the contents of ACCA, X, or M (prior to the NEG operation) is $00.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
NEGA	INH (A)	40		3
NEGX	INH (X)	50		3
NEG (opr)	DIR	30	dd	5
NEG, X	IX	70		5
NEG (opr),X	IX1	60	ff	6

NOP No Operation NOP

Description: This is a single-byte instruction that causes only the program counter to be incremented. No other registers are affected.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
NOP	INH	9D		2

Operation: ACCA ← (ACCA) + (M)

Description: Performs the logical inclusive-OR between the contents of ACCA and the contents of M and places the result in ACCA. Each bit of ACCA after the operation will be the logical inclusive-OR of the corresponding bits of M and of ACCA before the operation.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
ORA (opr)	IMM	AA	ii	2
ORA (opr)	DIR	BA	dd	3
ORA (opr)	EXT	CA	hh ll	4
ORA,X	IX	FA		3
ORA (opr),X	IX1	EA	ff	4
ORA (opr),X	IX2	DA	ee ff	5

ROL Rotate Left ROL

Operation: $\boxed{C} \leftarrow \boxed{\overleftarrow{b7 - - - - - - b0}} \leftarrow \boxed{C}$

Description: Shifts all bits of ACCA, X, or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the MSB of ACCA, X, or M. The rotate instructions include the carry bit to allow extension of the shift and rotate operations to multiple bytes. For example, to shift a 24-bit value left one bit, the sequence {ASL LOW, ROL MID, ROL HIGH} could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is \$00; cleared otherwise.

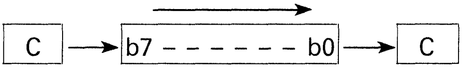
C b7
Set if, before the rotate, the MSB of ACCA or M was set; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
ROLA	INH (A)	49		3
ROLX	INH (X)	59		3
ROL (opr)	DIR	39	dd	5
ROL, X	IX	79		5
ROL (opr),X	IX1	69	ff	6

Rotate Right

Operation:



Description: Shift all bits of ACCA, X, or M one place to the right. Bit 7 is loaded from the C bit. The rotate operations include the carry bit to allow extension of the shift and rotate operations to multiple bytes. For example, to shift a 24-bit value right one bit, the sequence {LSR HIGH, ROR MID, ROR LOW} could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if result is $00; cleared otherwise.

C b0
Set if, before the rotate, the LSB of ACCA, X, or M was set; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
RORA	INH (A)	46		3
RORX	INH (X)	56		3
ROR (opr)	DIR	36	dd	5
ROR, X	IX	76		5
ROR (opr),X	IX1	66	ff	6

Reset Stack Pointer

Operation: SP ← $00FF

Description: Resets the stack pointer to the top of the stack.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
RSP	INH	9C	2

RTI Return from Interrupt RTI

Operation:

SP ← (SP)+$0001; ↑ CCR Restore CCR from stack
SP ← (SP)+$0001; ↑ ACCA Restore ACCA from stack
SP ← (SP)+$0001; ↑ X Restore X from stack
SP ← (SP)+$0001; ↑ PCH Restore PCH from stack
SP ← (SP)+$0001; ↑ PCL Restore PCL from stack

Description: The condition codes, accumulator, the index register, and the program counter are restored to the state previously saved on the stack. The I-bit will be reset if the corresponding bit stored on the stack is zero.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	↕	↕	↕	↕	↕

Set or cleared according the the byte pulled from the stack.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)	HCMOS Cycles
RTI	INH	80		9

RTS　Return from Subroutine　RTS

Operation: SP ← (SP)+$0001; ↑ PCH　Restore PCH from stack
SP ← (SP)+$0001; ↑ PCL　Restore PCL from stack

Description: The stack pointer is incremented by one. The contents of the byte of memory that is pointed to by the stack pointer is loaded into the high-order byte of the program counter. The stack pointer is again incremented by one. The contents of the byte of memory at the address now contained in the stack pointer is loaded into the low-order 8 bits of the program counter.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
RTS	INH	81	6

Subtract with Carry

Operation: ACCA ← (ACCA) − (M) − (C)

Description: Subtracts the contents of M and the contents of C from the contents of ACCA and places the result in ACCA.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N R7
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise.

C $\overline{A7} \cdot M7 + M7 \cdot R7 + R7 \cdot \overline{A7}$
Set if the absolute value of the contents of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
SBC (opr)	IMM	A2	ii		2
SBC (opr)	DIR	B2	dd		3
SBC (opr)	EXT	C2	hh	ll	4
SBC,X	IX	F2			3
SBC (opr),X	IX1	E2	ff		4
SBC (opr),X	IX2	D2	ee	ff	5

SEC Set Carry SEC

Operation: C bit ← 1

Description: Sets the C bit in the CCR. SEC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	1

C 1
Set

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
SEC	INH	99	2

Set Interrupt Mask

Operation: I bit ← 1

Description: Sets the interrupt mask bit in the CCR. The microprocessor is inhibited from servicing interrupts while the I bit is set.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	1	—	—	—

I 1
Set

Source Forms, Addressing Modes, Machine Code, and Cycles

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)	HCMOS Cycles
SEI	INH	9B		2

STA Store Accumulator STA

Operation: M ← (ACCA)

Description: Stores the contents of ACCA in memory. The contents of ACCA remain unchanged.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N A7
Set if MSB of result is set; cleared otherwise.

Z $\overline{A7} \bullet \overline{A6} \bullet \overline{A5} \bullet \overline{A4} \bullet \overline{A3} \bullet \overline{A2} \bullet \overline{A1} \bullet \overline{A0}$
Set if result is $00; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
STA (opr)	DIR	B7	dd		4
STA (opr)	EXT	C7	hh	ll	5
STA, X	IX	F7			4
STA (opr), X	IX1	E7	ff		5
STA (opr),X	IX2	D7	ee	ff	6

STOP — Enable $\overline{\text{IRQ}}$, Stop Oscillator — STOP

Description: Reduces power consumption by eliminating all dynamic power dissipation. This results in: 1) timer prescaler cleared, 2) timer interrupts disabled, 3) timer interrupt flag cleared, 4) external interrupt request enabled, and 5) oscillator inhibited.

When the $\overline{\text{RESET}}$ or $\overline{\text{IRQ}}$ input goes low, the oscillator is enabled, a delay of 1920 processor clock cycles is initiated allowing the oscillator to stabilize, the interrupt request vector or reset vector is fetched, and the service routine is executed, depending on which signal was applied.

External interrupts are enabled following the STOP command.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	0	—	—	—

I 0
Cleared

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
STOP	INH	8E	2

STX Store Index Register X STX

Operation: M ← (X)

Description: Stores the contents of X in memory. The contents of X remain unchanged.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N X7
Set if MSB of result is set; cleared otherwise.

Z $\overline{X7} \bullet \overline{X6} \bullet \overline{X5} \bullet \overline{X4} \bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$
Set if result is $00; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Machine Code Operand(s)		HCMOS Cycles
STX (opr)	DIR	BF	dd		4
STX (opr)	EXT	CF	hh	ll	5
STX, X	IX	FF			4
STX (opr), X	IX1	EF	ff		5
STX (opr),X	IX2	DF	ee	ff	6

SUB Subtract SUB

Operation: ACCA ← (ACCA) − (M)

Description: Subtracts the contents of M from the contents of ACCA and places the result in ACCA.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	↕

N $R7$
Set if MSB of result is set; cleared otherwise.

Z $\overline{R7} \cdot \overline{R6} \cdot \overline{R5} \cdot \overline{R4} \cdot \overline{R3} \cdot \overline{R2} \cdot \overline{R1} \cdot \overline{R0}$
Set if result is $00; cleared otherwise.

C $\overline{A7} \cdot M7 + M7 \cdot R7 + R7 \cdot \overline{A7}$
Set if the abolsute value of the contents of memory plus the previous carry is larger than the absolute value of the accumulator; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode	Operand(s)		HCMOS Cycles
SUB (opr)	IMM	A0	ii		2
SUB (opr)	DIR	B0	dd		3
SUB (opr)	EXT	C0	hh	ll	4
SUB,X	IX	F0			3
SUB (opr),X	IX1	E0	ff		4
SUB (opr),X	IX2	D0	ee	ff	5

Software Interrupt

Operation:

PC ← (PC) + $0001	Advance PC to return address
↓ (PCL); SP ← (SP) − $0001	Push low-order return address onto stack
↓ (PCH); SP ← (SP) − $0001	Push high-order return address onto stack
↓ (X); SP ← (SP) − $0001	Push index register onto stack
↓ (ACCA); SP ← (SP) − $0001	Push accumulator onto stack
↓ (CCR); SP ← (SP) − $0001	Push CCR onto stack
I bit ← 1	
PCH ← ($xFFC)	Vector fetch (x = 1 or 3 depending on
PCL ← ($xFFD)	HC05 version)

Description: The program counter is incremented by one. The program counter, index register, and accumulator are pushed onto the stack. The CCR bits are then pushed onto the stack, with bits H, I, N, Z, and C going into bit positions 4–0 and bit positions 7, 6, and 5 containing ones. The stack pointer is decremented by one after each byte of data is stored on the stack. The interrupt mask bit is then set. The program counter is then loaded with the address stored in the SWI vector (located at memory locations n–0002 and n–0003, where n is the address corresponding to a high state on all lines of the address bus). The address of the SWI vector can be expressed as $xFFC:$xFFD, where x is 1 or 3 depending on the version of 68HC05 being used. This instruction is not maskable by the I bit.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	1	—	—	—

I 1
Set

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
SWI	INH	83	10

TAX Transfer Accumulator to Index Register TAX

Operation: X ← (ACCA)

Description: Loads the index register with the contents of the accumulator. The contents of the accumulator are unchanged.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
TAX	INH	97	2

TST — Test for Negative or Zero — TST

Operation: (ACCA) − $00 **or:** (X) − $00 **or:** (M) − $00

Description: Sets the condition codes N and Z according to the contents of ACCA, X, or M. The contents of ACCA, X, and M are not altered.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	↕	↕	—

N M7
Set if the MSB of the contents of ACCA, X, or M is set; cleared otherwise.

Z $\overline{M7} \bullet \overline{M6} \bullet \overline{M5} \bullet \overline{M4} \bullet \overline{M3} \bullet \overline{M2} \bullet \overline{M1} \bullet \overline{M0}$
Set if the contents of ACCA, X, or M is $00; cleared otherwise.

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code		HCMOS Cycles
		Opcode	Operand(s)	
TSTA	INH (A)	4D		3
TSTX	INH (X)	5D		3
TST (opr)	DIR	3D	dd	4
TST, X	IX	7D		4
TST (opr),X	IX1	6D	ff	5

TXA Transfer Index Register to Accumulator TXA

Operation: ACCA ← (X)

Description: Loads the accumulator with the contents of the index register. The contents of the index register are not altered.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	—	—	—	—

None affected

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
TXA	INH	9F	2

WAIT Enable Interrupt, Stop Processor WAIT

Description: Reduces power consumption by eliminating most dynamic power dissipation. The timer, the timer prescaler, and the on-chip peripherals continue to operate because they are potential sources of an interrupt. Wait causes enabling of interrupts by clearing the I bit in the CCR and stops clocking of processor circuits.

Interrupts from on-chip peripherals may be enabled or disabled by local control bits prior to execution of the WAIT instruction.

When the $\overline{\text{RESET}}$ or $\overline{\text{IRQ}}$ input goes low or when any on-chip system requests interrupt service, the processor clocks are enabled, and the reset, IRQ, or other interrupt service request is processed.

Condition Codes and Boolean Formulae:

			H	I	N	Z	C
1	1	1	—	0	—	—	—

I 0
Cleared

Source Forms, Addressing Modes, Machine Code, and Cycles:

Source Forms	Addressing Mode	Machine Code Opcode Operand(s)	HCMOS Cycles
WAIT	INH	8F	2

A

APPENDIX B
REVIEW QUESTIONS

The 50 review questions presented are based directly on the text of this applications guide. These review questions are repeated with the proper answers, indicating the portion of text from which the information was obtained.

1. The instruction set of a CPU is
   ○ A. a software program written by an end user.
   ○ B. the same for all computers.
   ○ C. determined by the wiring within the CPU.
   ○ D. the data sheet for a microprocessor.

2. Which numbering system offers the best compromise between the needs of a CPU and those of a human?
   ○ A. Binary
   ○ B. Octal
   ○ C. Decimal
   ○ D. Hexadecimal

3. A specific 8-bit value in a computer memory can mean different things depending on its context. The value could be a number, a code representing an alphabetic character, a code for an instruction (opcode), etc. The hexadecimal value $42 could be interpreted by an MC68HC705C8 to mean any of the following things except one. Choose the one answer which is **not** likely to be a correct interpretation of the value $42.
   ○ A. The opcode for the MUL (multiply) instruction.
   ○ B. The decimal value 66.
   ○ C. The address of an on-chip control register.
   ○ D. The letter "B".

4. Which of the following items requires the most memory bits?
   ○ A. The BCD representation of 125.
   ○ B. The binary representation of 254.
   ○ C. The ASCII representation of the letter "A".
   ○ D. The binary equivalent of the octal number 75_8.

5. How many 8-bit memory locations would be needed to hold the ASCII representation of the name "FRED"?
○ A. 16
○ B. 4
○ C. 7
○ D. 2

6. Which of these CPU registers in the MC68HC705C8 contains the most bits?
○ A. The accumulator (A)
○ B. The index register (X)
○ C. The condition code register (CCR)
○ D. The program counter (PC)

7. Which CPU register in the MC68HC705C8 would most likely point to the next instruction that the CPU will execute?
○ A. The accumulator (A)
○ B. The index register (X)
○ C. The stack pointer (SP)
○ D. The program counter (PC)

8. During execution of a subroutine, where would the CPU save the return address? All except one of the following address pairs is incorrect due to improper memory type or address.
○ A. $1FFE,1FFF
○ B. $00EC,00ED
○ C. $00AE,00AF
○ D. $015E,015F

9. How many different opcodes correspond to the LDA (load accumulator) instruction?
○ A. 1
○ B. 3
○ C. 6
○ D. 16

10. In the following partial listing, what 8-bit value or code is present in memory location $0193?

```
018c          TIME   EQU  *      Update Time-of-day
018c 3d a2           TST  TIC    Check for TIC=zero
018e 26 38           BNE  XTIME  If not; just exit
0190 3c a3           INC  SEC    SEC=SEC+1
0192 a6 3c           LDA  #60
0194 b1 a3           CMP  SEC    Did SEC -> 60 ?
```

○ A. $A2
○ B. $3C
○ C. $93
○ D. $01

11. The following instruction reads the current value of the 8-bit variable "TIC" and internally tests for a negative or zero value. At what physical address is the variable "TIC" located?

```
018c 3d a2           TST  TIC    Check for TIC=zero
```

○ A. $01A2
○ B. $018D
○ C. $3DA2
○ D. $00A2

12. After executing the following sequence of instructions, what value will be in the accumulator?

```
BEGIN  LDA  #$80
       BPL  LABEL
       INCA
LABEL  DECA
       DECA
```

○ A. $7E
○ B. $7F
○ C. $80
○ D. $81

13. After executing the following instruction sequence from "START" to "END", what value will be in memory location $00FF?

```
0100 9c       START  RSP         Reset SP to $00FF
0101 cd 02 00        JSR  SUB    Call SUB
0104 cd 02 00        JSR  SUB    Call SUB again
0107 9d       END    NOP         Done
"    "    "   "       "           "
0200 81       SUB    RTS         Just Return
```

○ A. $00
○ B. $01
○ C. $04
○ D. $07

14. What frequency crystal would be used on an MC68HC705C8 to get a 500-ns internal processor clock?
○ A. 1.0 MHz
○ B. 2.0 MHz
○ C. 4.0 MHz
○ D. 8.0 MHz

15. For an MC68HC705C8 with a 4.0-MHz crystal, what amount of time corresponds to a single count of the 16-bit timer?
○ A. 500 ns
○ B. 1.0 μs
○ C. 2.0 μs
○ D. 4.0 μs

16. For an MC68HC705C8 with a 4.0-MHz crystal, what is the fastest baud rate available for the SCI (UART-type serial interface)?
○ A. 131.072 kbaud
○ B. 125 kbaud
○ C. 19.2 kbaud
○ D. 9600 baud

17. For an MC68HC705C8 with a 4.0-MHz crystal, what is the fastest master mode bit rate available for the SPI (synchronous serial peripheral interface)?
○ A. 1 Mbit/sec
○ B. 500 kbits/sec
○ C. 250 kbits/sec
○ D. 125 kbits/sec

18. How many bit times are there in one SCI character frame?
   ○ A. 8
   ○ B. 9
   ○ C. 10
   ○ D. 10 or 11

19. To assure an orderly startup, reset forces the CPU to begin executing instructions in a predictable, repeatable way. Which of the following statements best describes how the CPU proceeds from reset?
   ○ A. The CPU fetches the instruction from $1FFF and executes it.
   ○ B. The CPU loads the program counter (PC) with the address $1FFE and begins executing instructions.
   ○ C. The CPU begins executing instructions starting at address $0000.
   ○ D. The CPU loads the program counter (PC) with the address stored at $1FFE,1FFF and then begins executing instructions starting at that address.

20. To change the SCI baud rate, what address would you write to?
   ○ A. $000D
   ○ B. $000E
   ○ C. $0D00
   ○ D. $100E

21. The half-carry bit (H) in the condition code register (CCR)
   ○ A. is used in rounding results of arithmetic operations.
   ○ B. indicates that the MSB of the accumulator is 1.
   ○ C. may be used to adjust the results of BCD add operations.
   ○ D. indicates a borrow occurred during a subtract operation.

22. In an MC68HC705C8 system which uses no interrupts, what is the maximum possible nesting depth for subroutines (without causing errors)? If one subroutine called a second subroutine, that would be a nesting depth of 2.
   ○ A. 2
   ○ B. 32
   ○ C. 64
   ○ D. 128

23. Which of the following on-chip systems would be used to detect problems with the oscillator?
   ○ A. Power-on reset
   ○ B. COP watchdog timer
   ○ C. Clock monitor
   ○ D. IRQ interrupt

24. In the following instruction sequence, a value is read into the accumulator. From what **address** is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003           SAM    EQU  $03     SAM equal an 8-bit value
1400           LARRY  EQU  $1400   LARRY equal a 16-bit value
0100                  ORG  $100    Set program starting point
0100 ae 02     TOP    LDX  #$02    Initialize index register
0102 a6 05            LDA  #$05    Read value into A
```

   ○ A. $0005
   ○ B. $0102
   ○ C. $0103
   ○ D. $a605

25. In the following instruction sequence, a value is read into the accumulator. From what **address** is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003           SAM    EQU  $03     SAM equal an 8-bit value
1400           LARRY  EQU  $1400   LARRY equal a 16-bit value
0100                  ORG  $100    Set program starting point
0100 ae 02     TOP    LDX  #$02    Initialize index register
0102 b6 05            LDA  $05     Read value into A
```

   ○ A. $0005
   ○ B. $0102
   ○ C. $0103
   ○ D. $b605

26. In the following instruction sequence, a value is read into the accumulator. From what **address** is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03      SAM equal an 8-bit value
1400            LARRY  EQU  $1400    LARRY equal a 16-bit value
0100                   ORG  $100     Set program starting point
0100 ae 02      TOP    LDX  #$02     Initialize index register
0102 c6 01 00          LDA  TOP      Read value into A
```

○ A. $0003
○ B. $0100
○ C. $0103
○ D. $0104

27. In the following instruction sequence, a value is read into the accumulator. From what **address** is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03      SAM equal an 8-bit value
1400            LARRY  EQU  $1400    LARRY equal a 16-bit value
0100                   ORG  $100     Set program starting point
0100 ae 02      TOP    LDX  #$02     Initialize index register
0102 f6                LDA  0,X      Read value into A
```

○ A. $0000
○ B. $0002
○ C. $0003
○ D. $0102

28. In the following instruction sequence, a value is read into the accumulator. From what **address** is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03      SAM equal an 8-bit value
1400            LARRY  EQU  $1400    LARRY equal a 16-bit value
0100                   ORG  $100     Set program starting point
0100 ae 02      TOP    LDX  #$02     Initialize index register
0102 e6 03             LDA  SAM,X    Read value into A
```

○ A. $0002
○ B. $0003
○ C. $0005
○ D. $0105

29. In the following instruction sequence, a value is read into the accumulator. From what **address** is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03       SAM equal an 8-bit value
1400            LARRY  EQU  $1400     LARRY equal a 16-bit value
0100                   ORG  $100      Set program starting point
0100 ae 02      TOP    LDX  #$02      Initialize index register
0102 d6 14 00          LDA  LARRY,X   Read value into A
```

○ A. $0002
○ B. $1400
○ C. $1402
○ D. $1600

30. After executing the following instruction sequence from "START" to "END," what value will be in the stack pointer (SP)?

```
0100 9c        START  RSP          Reset SP to $00FF
0101 cd 02 00         JSR   SUB    Call SUB
0104 cd 02 00         JSR   SUB    Call SUB again
0107 9d        END    NOP          Done
  "    "    "    "      "              "
0200 81        SUB    RTS          Just Return
```

○ A. $0200
○ B. $00FB
○ C. $00FD
○ D. $00FF

31. A microcontroller is
○ A. the CPU part of a digital binary computer.
○ B. the same thing as a microprocessor.
○ C. any system that includes an MCU integrated circuit.
○ D. a computer system including a CPU, memory, and peripherals on a single I.C.

32. After executing the following instruction sequence from "TOP" to "BOT", what values will be in locations $00A0 and $00A1, respectively?

```
0100 a6 f3     TOP     LDA   #%11110011 Initial value
0102 b7 a0             STA   $A0        For $00A0
0104 a6 81             LDA   #%10000001 Initial value
0106 b7 a1             STA   $A1        For $00A1
0108 38 a1             ASL   $A1        Comment left off
010a 39 a0             ROL   $A0        intentionally
010c 38 a1             ASL   $A1
010e 39 a0             ROL   $A0
0110 9d        BOT     NOP
```

○ A. $00A0;00A1 = 11110011 10000001
○ B. $00A0;00A1 = 11001100 00000100
○ C. $00A0;00A1 = 11001110 00000111
○ D. $00A0;00A1 = 11001110 00000100

Refer to the following four program listings to answer questions 33 through 38. These programs demonstrate four different ways to generate pulses at port A bit 0 of an MC68HC705C8. All four programs assume that port A has been configured as outputs by the data direction register (DDRA) equal $FF.

```
0100 a6 01     PROG1  LDA   #$01     [2] Pattern for bit 0 high
0102 b7 00            STA   $00      [4] Write to port A
0104 a6 00            LDA   #$00     [2] Pattern for bit 0 low
0106 b7 00            STA   $00      [4] Write to port A
0108 20 f6            BRA   PROG1    [3] Repeat loop continuously
```

```
0100 10 00     PROG2  BSET 0,$00     [5] Set port A bit 0
0102 11 00            BCLR 0,$00     [5] Clear port A bit 0
0104 20 fa            BRA   PROG2    [3] Repeat loop continuously
```

```
0100 a6 01     PROG3  LDA   #$01     [2] Pattern for bit 0 high
0102 5f               CLRX           [3] Pattern for bit 0 low
0103 b7 00     LOOP3  STA   $00      [4] Write to port A
0105 bf 00            STX   $00      [4] Write to port A
0107 20 fa            BRA   LOOP3    [3] Repeat loop continuously
```

```
0100 b6 00     PROG4  LDA   $00      [3] Read present port A data
0102 a8 01            EOR   #$01     [2] Form new port A pattern
0104 b7 00            STA   $00      [4] Write to port A
0106 20 f8            BRA   PROG4    [3] Repeat loop continuously
```

33. Which of the four programs requires the fewest bytes of program memory?

○ A. PROG1
○ B. PROG2
○ C. PROG3
○ D. PROG4

34. Which of the four programs produces the shortest pulse width (logic one at the pin)?

○ A. PROG1
○ B. PROG2
○ C. PROG3
○ D. PROG4

35. Which of the four programs produces the longest period?

○ A. PROG1
○ B. PROG2
○ C. PROG3
○ D. PROG4

36. Sometimes it is important to change the level on a pin without disturbing values in the CPU accumulator and other CPU registers. Which of the four programs uses no CPU registers other than the program counter (PC)?

○ A. PROG1
○ B. PROG2
○ C. PROG3
○ D. PROG4

37. Which of the four programs produces a square wave (equal high and low times)?

○ A. PROG1
○ B. PROG2
○ C. PROG3
○ D. PROG4

38. Some instructions affect only a single bit in a memory location while others affect all bits in a memory location. Which two of the four programs **do not** make any assumptions about other bits in port A?
○ A. PROG1 & PROG2
○ B. PROG2 & PROG4
○ C. PROG3 & PROG4
○ D. PROG4 & PROG1

39. On an MC68HC705C8, which of the following pins is an input-only pin?
○ A. RESET
○ B. Port D bit 4/SCK
○ C. Port D bit 7
○ D. Port A bit 7

40. What does the following sequence of instructions do?

```
0100 a6 08    START  LDA  #$08  Comments left off intentionally
0102 b7 1e           STA  $1E
0104 8e              STOP
```

○ A. Reset the COP watchdog timer and return to normal program.
○ B. Force a hardware RESET.
○ C. Store a value $08 in RAM and stop processing.
○ D. Enables the clock monitor and the COP watchdog timer.

41. For the four following addresses, which one would **not** allow you to read back an arbitrary value which you just wrote to that address?
○ A. $0004
○ B. $0050
○ C. $00FF
○ D. $1000

42. For an MC68HC705C8, which of the four following addresses would be the **best** address to store a product serial number **and** a variable which changed once a second? Refer to the memory map on page 3-11 of the applications guide.
○ A. $0000
○ B. $002F
○ C. $00FF
○ D. $015F

43. If you discovered an incorrect value in a memory location as you were starting volume production, which of the following memory types would require the longest time to correct the error?
○ A. RAM
○ B. ROM
○ C. EPROM
○ D. EEPROM

44. A microcontroller includes
○ A. a central processor unit (CPU).
○ B. memory.
○ C. I/O devices.
○ D. all of the above.

45. A central processor unit (CPU)
○ A. is part of a microcontroller (MCU).
○ B. is a complete computer system.
○ C. contains memory and I/O devices.
○ D. contains an MCU.

46. A memory is said to be volatile if it forgets its contents when power is removed for long periods of time. Which of the following memory types is volatile?
○ A. ROM
○ B. RAM
○ C. EPROM
○ D. EEPROM

47. An EPROM memory is normally erased by
○ A. software instructions.
○ B. infrared light.
○ C. ultraviolet light.
○ D. application of high voltage.

48. To program the OPTION register on the MC68HC705C8
○ A. program all bits as if they were EPROM.
○ B. program all bits as if they were RAM.
○ C. program one bit like RAM and the rest of the bits as if they were EPROM.
○ D. program one bit like EPROM and the rest of the bits as if they were RAM.

49. In the MC68HC705C8, bit manipulation instructions (BSET and BCLR)
   - ○ A. can be used to manipulate any on-chip I/O register or RAM location in the $0000 through $00FF area of memory.
   - ○ B. can be used to manipulate any location in the 8K-byte memory map.
   - ○ C. can be used only with indexed addressing modes.
   - ○ D. can be used to manipulate any on-chip RAM location.

50. Which of the following statements best describes what happens during an SPI data transfer between two MC68HC705C8 MCUs?
   - ○ A. A slave device transfers an 8-bit character to a master device.
   - ○ B. A master device transfers an 8-bit character to a slave device.
   - ○ C. A master and a slave exchange 8-bit data characters.
   - ○ D. A master device sends a start bit, 8 data bits, and a stop bit to a slave.

REVIEW QUESTIONS, ANSWERS, AND EXPLANATIONS

The questions that seem to give the most trouble are 40, 35, and 13 in that order. The problem on 35 is that it is a tricky question. The loop in PROG4 must be executed **twice** to make one period on the port pin. On 40, some persons who got the wrong answer seemed to be tricked by the indirect nature of this operation and chose D, thinking it was the closest thing to a correct answer. Almost all those who got 35 wrong chose A, which has the longest loop time but not the longest period. The majority of those who missed 13 seemed to think that the RAM locations in the stack are cleared as values are recovered from the stack during a return from subroutine — this assumption is incorrect. A few others got the stacking order reversed. The key to getting 13 right was to play computer very carefully.

1. The instruction set of a CPU is
   ○ A. a software program written by an end user.
   ○ B. the same for all computers.
   ☞ C. determined by the wiring within the CPU. (see p. 2-2 2nd ¶ and p. 2-4 last ¶)
   ○ D. the data sheet for a microprocessor.

2. Which numbering system offers the best compromise between the needs of a CPU and those of a human?
   ○ A. Binary
   ○ B. Octal
   ○ C. Decimal
   ☞ D. Hexadecimal

   See p. 2-2 last ¶ and p. 2-5. A few engineers who were around in the days of the PDP-8 or work a lot with minicomputers that still carry on the octal tradition may argue about this answer. The text on p. 2-5 and modern microcontroller data sheets explain why hexadecimal is the best choice.

3. A specific 8-bit value in a computer memory can mean different things depending on its context. The value could be a number, a code representing an alphabetic character, a code for an instruction (opcode), etc. The hexadecimal value $42 could be interpreted by an MC68HC705C8 to mean any of the following things except one. Choose the one answer which is **not** likely to be a correct interpretation of the value $42.
   ○ A. The opcode for the MUL (multiply) instruction. (see p. A-47)
   ○ B. The decimal value 66. (see Table 2-1 p. 2-3)
   ☞ C. The address of an on-chip control register.
   ○ D. The letter "B". (see Table 3-10 p. 3-67)

   By elimination, the correct response is answer C. Looking at the memory map (Figure 3-5 p. 3-11) you would find that address $42 is a RAM or PROM location; whereas, all on-chip control registers (except OPTION at $1FDF) are in the area from $0000 to $001F.

4. Which of the following items requires the most memory bits?
   ☞ A. The BCD representation of 125. (0001 0010 0101 or 12 bits)
   ○ B. The binary representation of 254. (1111 1110 or 8 bits)
   ○ C. The ASCII representation of the letter "A". (1000001 or 0100 0001, 7 or 8 bits)
   ○ D. The binary equivalent of the octal number 75_8. (111 101 or 6 bits)

   See pp. 2-2 through 2-6.

5. How many 8-bit memory locations would be needed to hold the ASCII representation of the name "FRED"?
   ○ A. 16
   ☞ B. 4 (See p. 2-4 3rd ¶. Each ASCII character takes one byte.)
   ○ C. 7
   ○ D. 2

6. Which of these CPU registers in the MC68HC705C8 contains the most bits?

   ○ A. The accumulator (A)
   ○ B. The index register (X)
   ○ C. The condition code register (CCR)
   ☞ D. The program counter (PC)

   See Figure 2-2 p. 2-8. The PC is 13 or 16 bits, depending on whether or not you count the upper three bits that are fixed. A and X are 8 bits each, and CCR is 5 or 8 (again depending on whether or not you count the upper three bits that are fixed).

7. Which CPU register in the MC68HC705C8 would most likely point to the next instruction that the CPU will execute?

   ○ A. The accumulator (A)
   ○ B. The index register (X)
   ○ C. The stack pointer (SP)
   ☞ D. The program counter (PC) (see p. 2-8 last ¶)

8. During execution of a subroutine, where would the CPU save the return address? All except one of the following address pairs is incorrect due to improper memory type or address.

   ○ A. $1FFE,1FFF
   ☞ B. $00EC,00ED
   ○ C. $00AE,00AF
   ○ D. $015E,015F

   See p. 3-16 last ¶ and **Section 2.6.1.4** beginning on p. 2-27 if you need help understanding subroutine calls.

9. How many different opcodes correspond to the LDA (load accumulator) instruction?

   ○ A. 1
   ○ B. 3
   ☞ C. 6 (see p. A-43 and p. 2-21 4th ¶)
   ○ D. 16

10. In the following partial listing, what 8-bit value or code is present in memory location $0193?

```
018c            TIME    EQU   *       Update Time-of-day
018c 3d a2              TST   TIC     Check for TIC=zero
018e 26 38              BNE   XTIME   If not; just exit
0190 3c a3              INC   SEC     SEC=SEC+1
0192 a6 3c              LDA   #60
0194 b1 a3              CMP   SEC     Did SEC -> 60 ?
```

○ A. $A2

☞ B. $3C (see **Sections 2.5.4** and **2.5.5** especially p. 2-22 4th ¶)

○ C. $93

○ D. $01

11. The following instruction reads the current value of the 8-bit variable "TIC" and internally tests for a negative or zero value. At what physical address is the variable "TIC" located?

```
018c 3d a2              TST   TIC     Check for TIC=zero
```

○ A. $01A2

○ B. $018D

○ C. $3DA2

☞ D. $00A2 (see p. 3-26 and p. A-65)

12. After executing the following sequence of instructions, what value will be in the accumulator?

```
BEGIN  LDA   #$80
       BPL   LABEL
       INCA
LABEL  DECA
       DECA
```

○ A. $7E

☞ B. $7F

○ C. $80

○ D. $81

The first instruction loads A with the immediate value $80 (which is negative). The second instruction will not branch because the N condition code flag is set. The CPU then increments A (to $81), then decrements A (to $80), and finally decrements A again (to $7F).

13. After executing the following instruction sequence from "START" to "END", what value will be in memory location $00FF?

```
0100 9c        START  RSP          Reset SP to $00FF
0101 cd 02 00         JSR   SUB    Call SUB
0104 cd 02 00         JSR   SUB    Call SUB again
0107 9d        END    NOP          Done
  "   "    "    "      "            "
0200 81        SUB    RTS          Just Return
```

○ A. $00
○ B. $01
○ C. $04
☞ D. $07

See **Section 2.6.2** especially p. 2-32 last ¶ and p. 2-33 6th ¶; see also p. 2-9 3rd ¶. In the course of executing this program segment, the CPU would call a subroutine (and store the return address at $00FF and $00FE), then return from the subroutine (which causes the return address to be recovered from the stack and the stack pointer to end up pointing at $00FF again). When the second subroutine call is executed, the return address (now $0107) is saved on the stack at $00FF and $00FE (with the $07 at $00FF). The second return from subroutine causes this return address to be read from the stack. Since no other value is stored to location $00FF during this program, $07 will still be there at the end of the sequence.

14. What frequency crystal would be used on an MC68HC705C8 to get a 500-ns internal processor clock?

○ A. 1.0 MHz
○ B. 2.0 MHz
☞ C. 4.0 MHz (see p. 3-6 7th ¶ or Figure 3-18 p. 3-59)
○ D. 8.0 MHz

15. For an MC68HC705C8 with a 4.0-MHz crystal, what amount of time corresponds to a single count of the 16-bit timer?

○ A. 500 ns
○ B. 1.0 μs
☞ C. 2.0 μs (see p. 3-82 upper right and p. 3-83 last ¶.)
○ D. 4.0 μs

16. For an MC68HC705C8 with a 4.0-MHz crystal, what is the fastest baud rate available for the SCI (UART-type serial interface)?
○ A. 131.072 kbaud
☞ B. 125 kbaud (see Table 3-8 p. 3-59 top entry in 4.0 column)
○ C. 19.2 kbaud
○ D. 9600 baud

17. For an MC68HC705C8 with a 4.0-MHz crystal, what is the fastest master mode bit rate available for the SPI (synchronous serial peripheral interface)?
☞ A. 1 Mbit/sec (see table on p. 3-75)
○ B. 500 kbits/sec
○ C. 250 kbits/sec
○ D. 125 kbits/sec
Only a master SPI device produces a serial clock. As a **slave**, the fastest bit rate the SPI can accept would be the crystal frequency divided by 2 (or 2 MHz for a 4-MHz crystal).

18. How many bit times are there in one SCI character frame?
○ A. 8
○ B. 9
○ C. 10
☞ D. 10 or 11 (see Figure 3-24 p. 3-64)

Don't forget to count the start and stop bit times.

19. To assure an orderly startup, reset forces the CPU to begin executing instructions in a predictable repeatable way. Which of the following statements best describes how the CPU proceeds from reset?
○ A. The CPU fetches the instruction from $1FFF and executes it.
○ B. The CPU loads the program counter (PC) register with the address $1FFE and begins executing instructions.
○ C. The CPU begins executing instructions starting at address $0000.
☞ D. The CPU loads the program counter (PC) with the address stored at $1FFE,1FFF and then begins executing instructions starting at that address.

See p. 3-18 item 14 and p. 2-8 last ¶. Think about the other three answers; you should see that they do not make sense.

20. To change the SCI baud rate, what address would you write to?

☞ A. $000D

○ B. $000E

○ C. $0D00

○ D. $100E

See memory map Figure 2-4 p. 2-12 or Figure 3-5 p. 3-11, or see Figure 3-17 p. 3-58. See also p. 2-13 4th ¶.

21. The half-carry bit (H) in the condition code register (CCR)

○ A. is used in rounding results of arithmetic operations. (describes the C bit)

○ B. indicates that the MSB of the accumulator is 1. (describes the N bit)

☞ C. may be used to adjust the results of BCD add operations.

○ D. indicates a borrow occurred during a subtract operation. (describes the C bit)

See p. 3-14 2nd ¶ and p. 2-6 3rd ¶.

22. In an MC68HC705C8 system which uses no interrupts, what is the maximum possible nesting depth for subroutines (without causing errors)? If one subroutine called a second subroutine, that would be a nesting depth of 2.

○ A. 2

☞ B. 32 (see p. 3-16 last ¶)

○ C. 64

○ D. 128

Remember that each subroutine call uses two 8-bit memory locations to store the return address.

23. Which of the following on-chip systems would be used to detect problems with the oscillator?

○ A. Power-on reset

○ B. COP watchdog timer

☞ C. Clock monitor (see p. 3-19 1st ¶ and p. 3-20 6th ¶)

○ D. IRQ interrupt

24. In the following instruction sequence, a value is read into the accumulator. From what address is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03       SAM equal an 8-bit value
1400            LARRY  EQU  $1400     LARRY equal a 16-bit value
0100                   ORG  $100      Set program starting point
0100 ae 02      TOP    LDX  #$02      Initialize index register
0102 a6 05             LDA  #$05      Read value into A
```

○ A. $0005
○ B. $0102
☞ C. $0103 (see immediate addressing mode p. 3-24)
○ D. $a605

25. In the following instruction sequence, a value is read into the accumulator. From what **address** is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03       SAM equal an 8-bit value
1400            LARRY  EQU  $1400     LARRY equal a 16-bit value
0100                   ORG  $100      Set program starting point
0100 ae 02      TOP    LDX  #$02      Initialize index register
0102 b6 05             LDA  $05       Read value into A
```

☞ A. $0005 (see direct addressing mode p. 3-26)
○ B. $0102
○ C. $0103
○ D. $b605

26. In the following instruction sequence, a value is read into the accumulator. From what address is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03       SAM equal an 8-bit value
1400            LARRY  EQU  $1400     LARRY equal a 16-bit value
0100                   ORG  $100      Set program starting point
0100 ae 02      TOP    LDX  #$02      Initialize index register
0102 c6 01 00          LDA  TOP       Read value into A
```

○ A. $0003
☞ B. $0100 (see extended addressing mode p. 3-25)
○ C. $0103
○ D. $0104

Although this instruction sequence has no practical use, it would assemble and function. The value loaded into A would be $AE (the opcode of the LDX-immediate instruction). If you were not familiar with the use of labels, you could have looked at the machine code C6 01 00. The C6 indicates the extended addressing mode variation of the LDA instruction and 0100 is the address of the operand that would be loaded into A.

27. In the following instruction sequence a value is read into the accumulator. From what address is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions)

```
0003              SAM    EQU  $03       SAM equal an 8-bit value
1400              LARRY  EQU  $1400     LARRY equal a 16-bit value
0100                     ORG  $100      Set program starting point
0100 ae 02        TOP    LDX  #$02      Initialize index register
0102 f6                  LDA  0,X       Read value into A
```

○ A. $0000
☞ B. $0002 (see indexed no-offset p. 3-28)
○ C. $0003
○ D. $0102

At the time the LDA 0,X instruction is executed, X contains $02 due to the previous instruction.

28. In the following instruction sequence a value is read into the accumulator. From what address is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003              SAM    EQU  $03       SAM equal an 8-bit value
1400              LARRY  EQU  $1400     LARRY equal a 16-bit value
0100                     ORG  $100      Set program starting point
0100 ae 02        TOP    LDX  #$02      Initialize index register
0102 e6 03               LDA  SAM,X     Read value into A
```

○ A. $0002
○ B. $0003
☞ C. $0005 (see indexed 8-bit offset p. 3-30)
○ D. $0105

Don't forget to add the current value of X ($02) to the value SAM ($03).

29. In the following instruction sequence, a value is read into the accumulator. From what address is this value being read? (It may be helpful to look at the machine code as well as the mnemonic instructions.)

```
0003            SAM    EQU  $03       SAM equal an 8-bit value
1400            LARRY  EQU  $1400     LARRY equal a 16-bit value
0100                   ORG  $100      Set program starting point
0100 ae 02      TOP    LDX  #$02      Initialize index register
0102 d6 14 00          LDA  LARRY,X   Read value into A
```

○ A. $0002
○ B. $1400
☞ C. $1402 (see indexed 16-bit offset p. 3-32)
○ D. $1600

Don't forget to add the current value of X ($02) to the value LARRY ($1400).

30. After executing the following instruction sequence from "START" to "END", what value will be in the stack pointer (SP)?

```
0100 9c         START  RSP          Reset SP to $00FF
0101 cd 02 00          JSR   SUB    Call SUB
0104 cd 02 00          JSR   SUB    Call SUB again
0107 9d         END    NOP          Done
  "   "   "      "      "            "
0200 81         SUB    RTS          Just Return
```

○ A. $0200
○ B. $00FB
○ C. $00FD
☞ D. $00FF

This is a variation of the exercise in **Section 2.6.1.4** and Figure 2-11. During execution the stack pointer will have the values FF-FE-FD-FE-FF-FE-FD-FE-FF.

31. A microcontroller is
○ A. the CPU part of a digital binary computer.
○ B. the same thing as a microprocessor.
○ C. any system that includes an MCU integrated circuit.
☞ D. a computer system including a CPU, memory, and peripherals on a single I.C.

See p. 2-1 1st sentence and p. 1-2 2nd ¶.

32. After executing the following instruction sequence from "TOP" to "BOT", what values will be in locations $00A0 and $00A1, respectively?

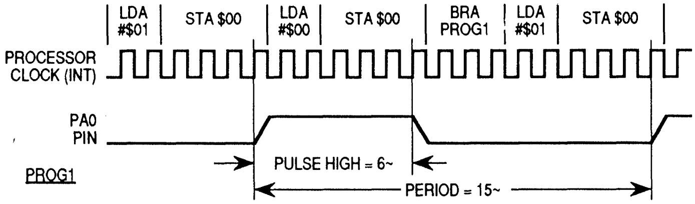
```
0100 a6 f3     TOP     LDA  #%11110011 Initial value
0102 b7 a0             STA  $A0        For $00A0
0104 a6 81             LDA  #%10000001 Initial value
0106 b7 a1             STA  $A1        For $00A1
0108 38 a1             ASL  $A1        Comment left off
010a 39 a0             ROL  $A0        intentionally
010c 38 a1             ASL  $A1
010e 39 a0             ROL  $A0
0110 9d        BOT     NOP
```

○ A. $00A0;00A1 = 11110011 10000001
○ B. $00A0;00A1 = 11001100 00000100
○ C. $00A0;00A1 = 11001110 00000111
☞ D. $00A0;00A1 = 11001110 00000100

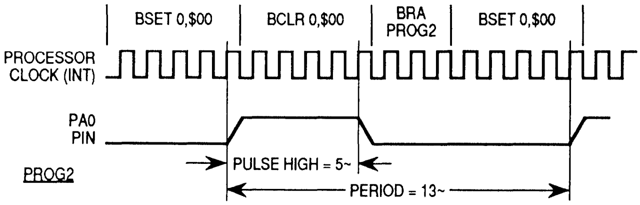
See ASL p. A-6 and ROL p. A-51. Play computer to see how this sequence works. This is a 16-bit version of the multibyte shift sequence described in the ROL instruction description.

Refer to the following four program listings to answer questions 33 through 38. These programs demonstrate four different ways to generate pulses at port A bit 0 of an MC68HC705C8. All four programs assume that port A has been configured as outputs by the data direction register (DDRA) equal $FF.

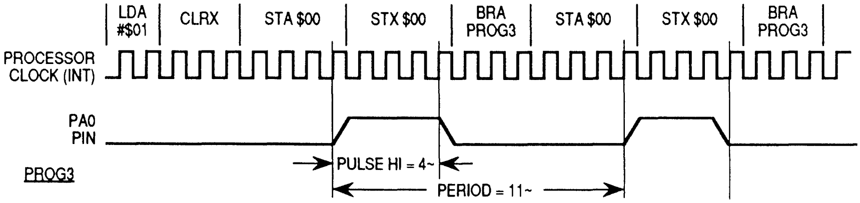
```
0100 a6 01     PROG1  LDA   #$01     [2] Pattern for bit 0 high
0102 b7 00            STA   $00      [4] Write to port A
0104 a6 00            LDA   #$00     [2] Pattern for bit 0 low
0106 b7 00            STA   $00      [4] Write to port A
0108 20 f6            BRA   PROG1    [3] Repeat loop continuously
```



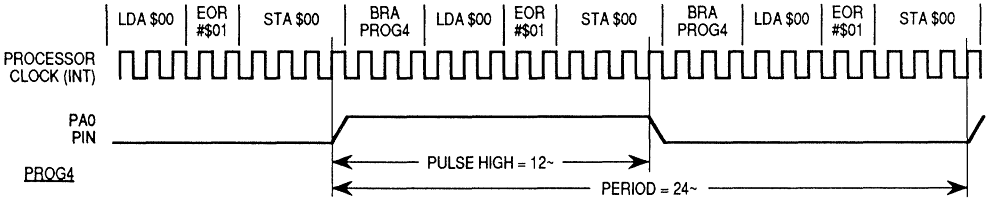
```
0100 10 00    PROG2  BSET 0,$00    [5] Set port A bit 0
0102 11 00           BCLR 0,$00    [5] Clear port A bit 0
0104 20 fa           BRA  PROG2    [3] Repeat loop continuously
```



```
0100 a6 01    PROG3  LDA   #$01    [2] Pattern for bit 0 high
0102 5f              CLRX          [3] Pattern for bit 0 low
0103 b7 00    LOOP3  STA   $00     [4] Write to port A
0105 bf 00           STX   $00     [4] Write to port A
0107 20 fa           BRA   LOOP3   [3] Repeat loop continuously
```



```
0100 b6 00    PROG4  LDA   $00     [3] Read present port A data
0102 a8 01           EOR   #$01    [2] Form new port A pattern
0104 b7 00           STA   $00     [4] Write to port A
0106 20 f8           BRA   PROG4   [3] Repeat loop continuously
```



33. Which of the four programs requires the fewest bytes of program memory?
    - ○ A. PROG1 (10)
    - ☞ B. PROG2 (6)
    - ○ C. PROG3 (9)
    - ○ D. PROG4 (8)

34. Which of the four programs produces the shortest pulse width (logic one at the pin)?
    - ○ A. PROG1 (6)
    - ○ B. PROG2 (5)
    - ☞ C. PROG3 (4)
    - ○ D. PROG4 (12)

35. Which of the four programs produces the longest period?
    - ○ A. PROG1 (15)
    - ○ B. PROG2 (13)
    - ○ C. PROG3 (11)
    - ☞ D. PROG4 (24) (Notice the loop executes twice to make a single period.)

36. Sometimes it is important to change the level on a pin without disturbing values in the CPU accumulator and other CPU registers. Which of the four programs uses no CPU registers other than the program counter (PC)?
    - ○ A. PROG1 (uses A)
    - ☞ B. PROG2 (BSET and BCLR use no CPU registers)
    - ○ C. PROG3 (uses A and X)
    - ○ D. PROG4 (uses A)

37. Which of the four programs produces a square wave (equal high and low times)?
    - ○ A. PROG1 (6/9)
    - ○ B. PROG2 (5/8)
    - ○ C. PROG3 (4/7)
    - ☞ D. PROG4 (12/12)

38. Some instructions affect only a single bit in a memory location; whereas, others affect all bits in a memory location. Which of the four programs **does not** make any assumptions about other bits in port A?
○ A. PROG1 & PROG2
☞ B. PROG2 & PROG4
○ C. PROG3 & PROG4
○ D. PROG4 & PROG1

Programs 1 and 3 force bits 7 through 1 of port A to zero; programs 2 and 4 affect only bit 0.

39. On an MC68HC705C8, which of the following pins is an input-only pin?
○ A. RESET
○ B. Port D bit 4/SCK
☞ C. Port D bit 7 (see Figure 3-1 p. 3-3)
○ D. Port A bit 7

This question was intended to emphasize that reset is **not** an input-only pin.

40. What does the following sequence of instructions do?

```
0100 a6 08    START  LDA  #$08  Comments left off intentionally
0102 b7 1e           STA  $1E
0104 8e              STOP
```

○ A. Reset the COP watchdog timer and return to normal program.
☞ B. Force a hardware RESET. (see p. 3-21 2nd ¶)
○ C. Store a value $08 in RAM and stop processing.
○ D. Enables the clock monitor and the COP watchdog timer.

This question was intended to show a way to force a reset with software, which may be useful in some applications. This question also reinforces important aspects of the clock monitor system and the STOP instruction.

41. For the four following addresses, which one would **not** allow you to read back an arbitrary value which you just wrote to that address?
○ A. $0004
○ B. $0050
○ C. $00FF
☞ D. $1000 (see Figure 3-5 p. 3-11)

$0050 and $00FF are RAM addresses and can obviously be read back after being written. $0004 is the data direction register for port A (see p. 3-52).

42. For an MC68HC705C8, which of the four following addresses would be the **best** address to store a product serial number **and** a variable which changed once a second? Refer to the memory map on p. 3-11 of the applications guide.
○ A. $0000
○ B. $002F
○ C. $00FF
☞ D. $015F (see description of RAM1 p. 3-98)

This question was intended to point out that the RAM1 control bit in the OPTION control register can be controlled by software to alternately enable RAM or PROM during normal operation. The result is that **both** the RAM **and** the PROM are usable, although software is required to choose which is active at any particular time. You could enable the PROM and program a serial number into location $015F before shipping a product. You could turn on the PROM during startup to read the serial number, then change RAM1 to enable the RAM to use the RAM located at $015F as the storage location for a software variable.

43. If you discovered an incorrect value in a memory location as you were starting volume production, which of the following memory types would require the longest time to correct the error?
○ A. RAM (RAM values can be changed in a single bus cycle or about 1 μs)
☞ B. ROM (ROM changes require several weeks because new parts must be manufactured.)
○ C. EPROM (EPROM takes several minutes of exposure to UV light to erase.)
○ D. EEPROM (EEPROM can be changed in tens of milliseconds. See p. 1-4 and p. 4-4 1st ¶).

44. A microcontroller includes
○ A. a central processor unit (CPU).
○ B. memory.
○ C. I/O devices.
☞ D. all of the above. (see **Section 1.1** p. 1-2)

45. A central processor unit (CPU)
☞ A. is part of a microcontroller (MCU). (see **Section 1.1** p. 1-2)
○ B. is a complete computer system.
○ C. contains memory and I/O devices.
○ D. contains an MCU.

46. A memory is said to be volatile if it forgets its contents when power is removed for long periods of time. Which of the following memory types is volatile?

○ A. ROM
☞ B. RAM
○ C. EPROM
○ D. EEPROM

See p. 1-4 2nd ¶ and p. 4-4 1st ¶.

47. An EPROM memory is normally erased by

○ A. software instructions.
○ B. infrared light.
☞ C. ultraviolet light. (see p. 1-4 5th ¶)
○ D. application of high voltage.

48. To program the OPTION register on the MC68HC705C8

○ A. program all bits as if they were EPROM.
○ B. program all bits as if they were RAM.
○ C. program one bit like RAM and the rest of the bits as if they were EPROM.
☞ D. program one bit like EPROM and the rest of the bits as if they were RAM. (see pp. 3-97 and 3-98)

49. In the MC68HC705C8, bit manipulation instructions (BSET and BCLR)

☞ A. can be used to manipulate any on-chip I/O register or RAM location in the $0000 through $00FF area of memory.
○ B. can be used to manipulate any location in the 8K-byte memory map.
○ C. can be used only with indexed addressing modes.
○ D. can be used to manipulate any on-chip RAM location.

See pp. A-9 and A-30.

50. Which of the following statements best describes what happens during an SPI data transfer between two MC68HC705C8 MCUs?

○ A. A slave device transfers an 8-bit character to a master device.
○ B. A master device transfers an 8-bit character to a slave device.
☞ C. A master and a slave exchange 8-bit data characters.
○ D. A master device sends a start bit, 8 data bits, and a stop bit to a slave.

See p. 3-71 2nd ¶.

A24686-1 PRINTED IN USA 6/91 GTE DIRECTORIES #14069 15,000 MCU YGACAA

Literature Distribution Centers:

USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.
EUROPE: Motorola Ltd.; European Literature Center; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.
ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.; P.O. Box 80300; Cheung Sha Wan Post Office; Kowloon Hong Kong.
JAPAN: Nippon Motorola Ltd.; 3-20-1 Minamiazabu, Minato-ku, Tokyo 106 Japan.

MOTOROLA