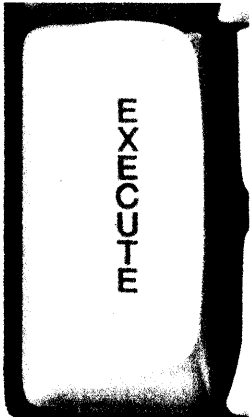
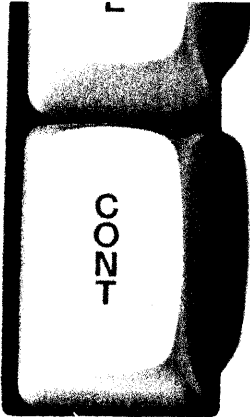


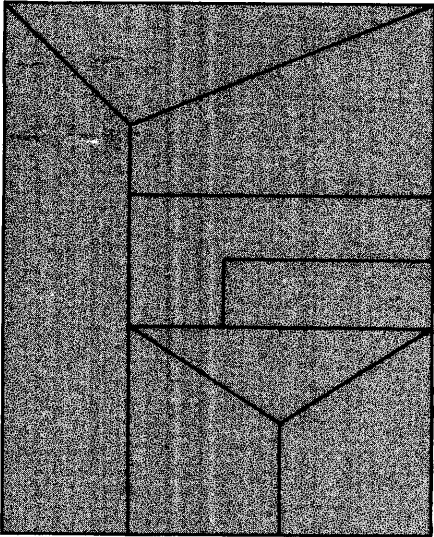


Pascal System Designer's Guide

```
CALL Build_array)
40 CALL Sort_array(N)
50 PRINT FNSum_array)
60 END
70 SUB Build_array(X)
80 ! X(*) is the array
90 ! N tells how many elements are assumed
```



```
done:=FALSE; btm:=0; top:=1;
FOR loop:=1 TO top DO alpha[loop]:=loop;
WRITELN('Type uppercase character');
READ(key); WRITELN;
WHILE NOT done DO
```



```
drive D:\;cat cat
for I=1 to 7;red "cat";
ent "Destination";
ent "Destination";
red "cat",A$;if len(A$)=7;
end
```



Warranty Statement

Hewlett-Packard makes no expressed or implied warranty of any kind, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, with regard to the program material contained herein. Hewlett-Packard shall not be liable for incidental or consequential damages in connection with, or arising out of, the furnishing, performance or use of this program material.

HP warrants that its software and firmware designated by HP for use with a CPU will execute its programming instructions when properly installed on that CPU. HP does not warrant that the operation of the CPU, software, or firmware will be uninterrupted or error free.

Use of this manual is restricted to this product only. Resale of the programs listed in their present form or with alterations, is expressly prohibited.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

Pascal 2.0 **System Designers' Guide** *for HP Series 200 Computers*

Manual Part No. 09826-90074

Microfiche No. 09826-99074

© Copyright Hewlett-Packard Company, 1983

This document refers to proprietary computer software which is protected by copyright. All rights are reserved. Copying or other reproduction of this program except for archival purposes is prohibited without the prior written consent of Hewlett-Packard Company.

Hewlett-Packard Desktop Computer Division
3404 East Harmony Road, Fort Collins, Colorado 80525



Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

March 1983...First Edition

Table of Contents

Chapter 1

Introduction

System Internals Documentation.....	1
Fair Warning.....	2
How to use this Documentation.....	4
Differences between Releases 1.0 and 2.0.....	5
System Distribution.....	6
The CTABLE Program.....	6
File System.....	6
Object Code Incompatibility.....	7
IO Drivers.....	8
New Peripheral Support.....	8
Miscellaneous.....	8
Software Tools Used for System Generation.....	9
Assembler and Librarian.....	9
Pascal Compiler.....	9
Memory Allocation of Variables.....	10

Chapter 2

Peripheral Configuration

Introduction.....	11
Definitions.....	11
Principles of Auto-Configuration.....	13
Alternate Directory Access Methods.....	15
Selecting the Primary DAM.....	16
Comparison of LIF and Workstation DAMs.....	16
File Interchange between WS1.0 and LIF.....	17
Supported Mass Storage Products.....	18
BASIC and Pascal File Interchange.....	19
Library Management.....	20
The System LIBRARY.....	21
The INTERFACE File.....	21
The IO File.....	22
The GRAPHICS File.....	22
Adding Modules to the System LIBRARY.....	23
Notes and Possible Problems.....	23
INITLIB and Memory-Resident Modules.....	24
Notes on Various Modules.....	27

Chapter 3

Modifying the Configuration

Introduction.....	29
General process.....	29
Commentary on the CTABLE Program.....	31
Modifying Module OPTIONS.....	32
Module CTR.....	35
Module BRSTUFF.....	36
Module SCANSTUFF.....	36
Discussion of the Main Body of CTABLE.....	36
An Example Configuration.....	38

Mass Storage Setup.....	38
Editing CTABLE.....	39
Compiling and Running CTABLE.....	41
Verifying the New Configuration.....	41
Making the New Configuration Permanent.....	42
Hard Disc Partitioning.....	43
Summary.....	44

Chapter 4

SRM Set-Up

Overview.....	45
Booting.....	45
The System Volume.....	45
SRM Initialization.....	47

Chapter 5

Programming with Files

File Naming Conventions.....	61
File Specifications and File Names.....	61
Syntax of a File Specification.....	61
Syntax of a Volume Identifier.....	62
Syntax of a Path Name.....	63
Syntax of File Names.....	64
File Types Derived From File Names.....	64
File Names (LIF Directory).....	65
File Names (WORKSTATION 1.0 Directory).....	66
File Names (SHARED RESOURCE MANAGER).....	66
File Size Specification.....	66
Operations.....	67
Pascal Primitive File Operations.....	67
File Position.....	69
The Buffer Variable.....	69
File States.....	69
Creating New Files.....	70
Disposing of Files.....	72
Opening Existing Files.....	72
Sequential File Operations.....	73
Direct Access (Random Access) Files.....	75
Textfile Input and Output.....	76
Declaring a TEXT File.....	76
Representations of a TEXT File.....	77
Formatted Input and Output.....	78
Reading a STRING or PAC from a Textfile.....	79
The Third Parameter.....	80
SRM Concurrent File Access.....	81
SRM Access Rights.....	83
Debugging Programs Which use Files.....	84

Chapter 6

The Booting Process

Introduction.....	85
Concepts of Linking and Loading.....	85
Overview of the Booting Process.....	87

How the Boot Files are Chosen.....	88
Memory Map Development.....	89
Summary of the Booting Process.....	100
The Pascal Kernel.....	101
Refresher on Pascal Modules.....	101
Modules in the Kernel.....	103
Digression on a Trick.....	105
Chapter 7	
The File System	
Introduction.....	107
Representation of File Variables.....	108
High-Level File Operations.....	109
The Access Methods.....	110
The Unit Table.....	113
The Transfer Methods.....	113
The Directory Access Methods.....	114
How the Access Method is Selected.....	115
Fields of an FIB.....	118
The Unitable.....	126
The Fields of a Unit Entry.....	127
Chapter 8	
File Support	
Introduction.....	133
Error Reporting by the File IO Subsystems.....	142
File System Exports.....	144
Chapter 9	
DAM's	
Reference Specification for DAM's.....	221
The Golden Rule for DAMs.....	222
Calling DAM's.....	222
The LIF Directory Access Method.....	231
Implementation of LIFMODULE.....	233
LIF Directory File Names.....	234
Routines within Procedure LIFDAM.....	235
Details on Various DAM Requests.....	240
Chapter 10	
File Operations	
Introduction.....	247
Filepack Examples.....	248
Writing your Own Command Interpreter.....	275
The Standard Command Interpreter.....	275
Creating a New Command Interpreter.....	276
Structure of the Command Interpreter.....	277
Sample Command Interpreter.....	278

Chapter 11

CPU Interrupt Handling

Introduction.....	281
Hooking in Your Own ISR.....	286
A Cautionary Note.....	287
Restrictions on Interrupt Service Routines.....	288
Error Conditions "Thrown Away".....	288
The "ISR in an ISR" Mistake.....	288

Chapter 12

The Keyboard

Introduction.....	291
Summary of Keyboard Capabilities.....	291
Keyboard Access with the File System.....	293
Echoing Read.....	293
Non-Echoing Read.....	294
The Beeper.....	295
Easy-to-Use Extensions.....	295
Avoiding "Hanging Reads".....	295
Timing with the System Clock.....	296
Using the System Clock and Calender.....	296
Remapping the Keyboard.....	297
Here is What You Want to Know.....	298
KBDHOOK Status Byte.....	299
The Large Keyboard.....	300
The Small Keyboard.....	301
An Example Keyboard Program.....	303
Commentary on the Example.....	306
Gritty Details of the Keyboard.....	307
About the Electronics.....	307
Keyboard Microcomputer.....	307
Clock.....	307
Non-Detachable Keyboard Scanning.....	307
Detachable Keyboard Scanning.....	308
Circuits Common to both Keyboards.....	309
The Rotary Pulse Generator (Knob).....	309
The Beeper.....	309
Protocol for Keyboard Handling.....	310
Communication Addresses.....	310
Interrupting the 68000.....	310
Sending a Command to the 8041.....	311
"Black Box" Description of Functions.....	311
Keyboard Command Processing.....	318
Processing an 8041 Service Request.....	320
Knob and Timer Details.....	321
The Keyboard at Power-up and Reset.....	322
Pascal Interface to the Keyboard.....	323

Chapter 13

Alpha and Graphics Displays

Introduction.....	325
Display Capabilities.....	326
9836 Alpha.....	326
9836 Graphics.....	326
9826 Alpha.....	327
9826 Graphics.....	327
9816 Alpha.....	328
9816 Graphics.....	328
Alpha Screen Driver Considerations.....	329
Controlling Character Attributes.....	329
The 6845 CRT Controller.....	330
Registers 10 and 11.....	330
Registers 12 and 13.....	331
Registers 14 and 15.....	332
Graphics Screen Driver Considerations.....	333
Pascal Access to the CRT.....	334
File System Operations.....	334
Scrolling.....	335
Lower-Level Access to the CRT.....	335
Cursor Motion.....	335
Interrogating the Dimensions of the CRT.....	336
Writing Directly to Screen Locations.....	336
Fiddling with the Typeahead Buffer.....	336
Turning the Screens On and Off.....	337
Dumping the Alpha or Graphic Screens.....	337

Chapter 14

The MISCINFO File

Introduction.....	339
The Listing.....	340

Chapter 15

Internal Disc Drives

Floppy Control Board.....	345
Theory of Operation.....	345
Status and Control Registers.....	348
On-Board RAM (256 byte buffer).....	351
Commands and Status.....	352
Type I Command Flags.....	352
Type II Command Flags.....	353
Type III Command Flags.....	354
Type IV Command Flags.....	354
Type I Commands.....	355
Type II Commands.....	356
Type III Commands.....	357
Type IV Commands.....	358
Status Information.....	358
Programming Considerations.....	360

Chapter 16

The Power-Fail Option

Introduction.....	361
Features.....	361
Power-Fail Behavior.....	361
Real-Time Clock.....	362
Non-Volatile RAM.....	362
Interface to the 68000.....	363
Commands to the Battery.....	364
Pascal Programming Interface.....	367

Chapter 17

Object Code Format

Introduction.....	369
Purposes of the Object Code Format.....	369
Definitions.....	369
Structure of a Library File.....	370
Library Directory.....	370
Module Directory.....	371
General Value or Address Record (GVR).....	373
Flags.....	374
Reference Pointer.....	375
How a GVR is evaluated.....	376
EXT Table (External Symbol Table).....	376
DEF Table (Definition Symbol Table).....	377
Define Source.....	378
TEXT Record.....	378
REF Tables.....	378
Miscellaneous Notes.....	379

Chapter 18

The Boot ROMs

Introduction.....	381
Overview.....	382
Boot Formats.....	383
ROM Headers.....	383
Boot Disc Formats.....	387
LIF System File Format.....	388
SDF Boot Area Format.....	391
UNIX (R) Boot Area Format.....	392
ROM/EEPROM Pseudo-Disc Format.....	392
SRM System Files.....	393
Default Mass Storage.....	394
CPU Board PROM.....	398
Machine Configuration.....	400
SYSFLAG.....	401
SYSFLAG2.....	402
BATTERY.....	402
CRTID, CRT Presence, Graphics Presence.....	403
Keyboard.....	406
NDRIVES.....	406
BOOT ROM Configuration and Revision.....	406

CPU State at Load.....	408
Read Interface and Secondary Loading.....	409
Flexible Disc Drivers.....	415
System Switching.....	422
REQ_BOOT.....	422
REQ_REBOOT.....	423
BOOT.....	423
CRTINIT.....	425
CRTCLEAR/CRTMSG.....	426
NMI_DECODE.....	428
CRASH.....	429
Character Table.....	429
High RAM Map.....	430
Low ROM Map - Exception Vectors.....	433
Using Boot ROM Routines from Pascal.....	435
Creating a Bootable System.....	437
Guidelines for System Creation.....	437
Rules for Using the Boot Command.....	438
An Example.....	439
Trap / Exception Vectors used in Pascal.....	440

Chapter 19

Device I/O

Introduction.....	441
The Hardware View.....	441
The Programmer View.....	442
General Architecture.....	444
Main Data Structures.....	447
ISC_TABLE.....	447
Driver Read/Write.....	450
Buffer Control Block.....	452
Driver Structure.....	454
High Level Routines.....	457
Execution Walkthrough.....	458
Power-Up.....	458
Stop Key.....	459
Program Compilation and Execution.....	459
Low Level Drivers.....	461
HP-IB.....	461
GPIO.....	462
DMA.....	462
Data Comm.....	463
I/O Examples.....	465
Using Special Buffers.....	465
Remote Console Driver.....	466
Removal of Drivers.....	493
Addition of a Driver.....	494
Modification of a Driver.....	498
End-of-Transfer Procedures.....	499
Interrupt Service Routine Procedures.....	502
HP-IB Interrupts.....	503
GPIO Interrupts.....	509
Serial Interrupts.....	514

Chapter 20
The DIO Bus

Introduction.....	523
Objectives.....	523
Designer's Responsibilities.....	523
Signal Terminology.....	524
System Elements.....	524
Bus Timing Background.....	526
Memory Map.....	528
Series 200 Memory Map.....	528
Registers.....	531
Data Transfers.....	534
Data Transfer Signals.....	534
Data Transfer Overview.....	535
Read Cycle Description.....	535
Write Cycle Description.....	538
Read-Modify-Write.....	541
Enable DTACK Timing.....	541
Direct Memory Access.....	544
DMA Signals.....	544
DMA Overview.....	544
DMA Output Cycle Description.....	545
DMA Input Cycle Description.....	547
DMA Speed Considerations.....	549
Terminating DMA Transfers.....	550
Bus Error Operation.....	551
The Bus Error Signal.....	551
Bus Timeouts.....	551
Interrupt Operation.....	553
Interrupt Signals.....	553
Interrupt Description.....	554
Utility Signals.....	555
Signals.....	555
Reset Operations.....	555
Function Code Signals.....	556
Electrical Specifications.....	557
Power Distribution and Grounding.....	557
Power Supply Tolerances.....	557
Power Requirements of Cards.....	558
On-Card Fuse Specifications.....	560
Signal Loading.....	560
Mechanical Specifications.....	562
Specifications for Cards.....	562
Card Cage Specifications.....	564
I/O Card Coverplate.....	565
Minimizing Electromagnetic Noise.....	565
PC-Board Layout Rules.....	566
Pin Assignments.....	566
Operation in the Bus Expander.....	568
Features of the Bus Expander.....	568
Operating Limitations With the Expander.....	568
Design Summary.....	569

I/O Card Design Guidelines.....	569
I/O Card Design Example.....	569
Design Qualification.....	575
Safety Compliance.....	575
Hardware Qualification.....	576
Software Qualification.....	577

Chapter 1

Introduction

System Internals Documentation

You are reading the *System Designers' Guide* to the HP Series 200 Computers. It is one part of the *System Internals Documentation*.

The Internals documentation includes:

- The System Designers' Guide
- Pascal 2.0 Source Code Listings (Volume I)
- Pascal 2.0 Source Code Listings (Volume II)

The source listings provide you with examples of the actual code used in the creation of the Pascal 2.0 Language System.

The *System Designers' Guide* is a collection of engineering notes and reference-specifications describing the software interface to the hardware systems supported by Pascal 2.0. Inside you will find detailed information covering a broad range of topics -- to quote one of the designers, "Enough information to make you dangerous."

Fair Warning

If you use the information in this document, you are writing hardware and operating-system dependent programs which will, by definition, be hard to transport to other computers or operating systems. The decision to do so, and the consequences, are your responsibility. Here are some suggestions and observations which may help protect your investment in HP products.

Sometimes the competitive need to innovate will force system designers into really unpleasant decisions which may invalidate code customers have written. Technologies are changing more rapidly and radically than ever before; no one is wise enough to foresee or design for every eventuality, and really big steps like transparent remote file access (the Shared Resource Manager) are bound to create transportability problems.

Even if your application is written in "vanilla" Pascal and has essentially no system dependencies, you might have to recompile it to move to a new release. (We try hard to avoid this!) But there is an HP Pascal language standard; we do our very best to stick to it in letter and in spirit.

If your application accesses modules of the OS and fiddles with system variables or calls system routines, there is more danger of creating a serious problem sometime in the future. However, the module interfacing techniques used to build this system give considerable protection as long as the program in question runs only under this operating system. So long as we aren't forced to change a module's interface, your code should upgrade freely; even if we must change an interface, you probably need only recompile. This is in sharp contrast to systems written in assembly language, which are often dependent on addresses which change from release to release.

If you write an assembly language routine which accesses system variables, using hard-coded displacements into the global area or some equally rigid arrangement, it is likely to create a hassle someday. We encourage programmers -- our own included -- to approach assembly language this way:

1. Write the whole application in Pascal first. Use system programming extensions if need be. Don't worry about speed, most people are amazed by the computational performance of the 68000.
2. If some part of the application is too slow, think carefully about the options to improve it. For instance, suppose the program repeatedly reads a voltmeter and seems to take longer than can be tolerated. If you are using the highest level of the IO Library, you're driving a luxury car. It is very easy, but it wallows around the curves. You might process the voltmeter readings as fixed-point numbers (scaled integers) instead of floating point, and do character IO directly by calling a lower level of the IO Library, thus avoiding some overhead.

Going to a lower level entry into the system is an option which doesn't exist with most interpreted systems. The advantage of this route is that by directly importing the lower-level modules, you let the Compiler take care of resolving the interfaces. If things move around, you need only recompile to adapt.

3. If you decide you must write assembly code, design your routines so that they operate only on parameters passed in, without side effects on variables in other modules. It is often wise to use the information in the Assembler chapter of the Pascal Language System User's Manual to make your assembly code look like code generated by the Compiler.

This warning is not intended to give you a warm, fuzzy feeling; it is intended to be fair. We have had enough requests for this information to believe that meets a need. But before diving in, be sure you can afford the swim.

How to use this Documentation

This documentation consists of three books: the one you are reading, a volume of system listings written in HP's system programming dialect of Pascal, and a volume of assembly language code which also includes a cross-reference of the Pascal portions.

Be aware that these source listings are proprietary material and are protected by copyright. They are provided for reference purposes only.

These listings are NOT complete -- they cover those parts of the system which we wanted to make accessible to customers, and suppress other parts. What is presented in detail is: IO drivers and underlying software support architecture; interrupt handling; object code format and the process of linking and loading programs; memory maps and development of the execution environment.

Other levels of the system are documented only at their interfaces. For instance, the file support level (routines called by the compiler) is documented by specifying the procedures which can be called, and what the stack should look like upon entry. This should simplify interfacing other compilers to the OS.

Low-level manipulations of files are performed by calls to "Directory Access Methods" (DAM's) and "Access Methods" (AM's). The architecture of this level is discussed, and there are detailed examples showing how to program the most important operations.

The purpose of this document is to tell you how to write programs which "get inside" the machine and make it do some very specialized things. The document is not primarily a hardware guide, although there is a goodly amount of material on the hardware. A typical reason for using the information published here might be to write and install a device driver for a non-HP IO interface card.

We did not concentrate on documenting the Series 200 family at the lowest (hardware) level primarily because we felt that most customers would be best served by building on the software base we have created. We believe you will be better off, for instance, using our disc drivers than trying to write your own. Ours were written by experts, and they protect you from ruining expensive disc drives. Moreover, we have provided a uniform interfacing structure. If new mass storage products are added and you are using the Pascal support structures, your programs should be able to use the new products right away. Another example is the HP-IB interface. It can be made to do some simple things fairly readily, but to explain all its idiosyncrasies and strange states would take more doing than seems justified by the requests we have had for information.

To use this material successfully, you must be a good Pascal programmer, acquainted with the concepts of system programming, and familiar with OS design principles in general. You should have read and understood the contents of

- *Pascal Language System User's Manual* (concentrate on modules, system programming language extensions)
- *Pascal Procedure Library User's Guide* (know the concepts of physical device IO)
- *MC68000 User's Manual* (know your computer)

Generally, this documentation has been written in a style that requires you to read it rather than just use it for reference.

Differences between Releases 1.0 and 2.0

This material describes the internal organization and specifications of release 2.0 of the Pascal language system for HP Series 200 desktop computers.

There are very substantial differences between the internal structures of the 1.0 and 2.0 releases, so this material is not a good guide to the innards of Pascal 1.0; to use it that way would be very misleading. Don't try.

The main differences between the releases are:

- Organization of Discs on which the System is Distributed
- Peripheral Configuration (CTABLE Program)
- File System
- Object Code Incompatibility
- IO Drivers
- New Peripheral Support
- Miscellaneous

Each of these issues is discussed in the following sections.

System Distribution

The Pascal 1.0 software was distributed on a set of four discs. The system library file contained the entire complement of IO, Graphics and OS interface modules. The system as booted up by the user contained IO driver software for all supported peripheral devices.

Pascal 2.0 is distributed on six discs. The system library file is almost empty, and the IO, Graphics and OS interface modules are supplied on a separate disc. The user can put just the ones he wants into his system library. The system as supplied contains IO driver software for the most common peripherals but not for all; this was done to conserve memory for the average user, since Pascal 2.0 supports many more peripheral devices. The less commonly needed drivers are supplied on a separate disc.

Consequently, before compiling or running programs which do device IO or graphics, the required modules should be added to the system library. Similarly, to configure a system to use certain peripherals, the Librarian must be used to install the required driver software.

Documentation is provided which explains how and when to install optional software into the system library and the Operating System.

The CTABLE Program

Pascal 2.0 scans interfaces for various peripherals and automatically configures itself. This is a considerable improvement over the 1.0 release. Auto-configuration is discussed in the next chapter.

In this regard, the most important change is that HP's Logical Interchange Format (LIF) directory structure is now the primary disc organization for Pascal 2.0, as opposed to the directory structure used by Pascal 1.0. This does not mean that Pascal 1.0 discs are inaccessible, or even that you need to convert them. Pascal 2.0 can be configured to used either directory format. See the next chapter for details.

File System

The Pascal 1.0 file system was only able to cleanly handle a single directory organization. We provided a library of routines to access Logical Interchange Format (LIF) discs, but they were not integrated into the file system.

As will be explained, the LIF library is not present in the system you just received; it is no longer necessary. The Lfiler (LIF Filer) is also unnecessary and has gone away, since the standard system Filer can now do the job. The 2.0 Filer is a completely new program, although its surface behavior is like the 1.0 Filer.

The 2.0 file system is completely reorganized. It has been broken into levels called File Support (FS), Directory Access Method (DAM), Access Method (AM) and Transfer Method (TM). This new organization allows the system to handle any number of different directory formats, and separates out the processing of each type of file structure which is supported. In fact, a customer can invent a new directory format or file type and bind it into the system so it can be used by all programs.

The Directory Access Methods now supported are: Workstation 1.0 compatible (same format as Pascal 1.0), HP Logical Interchange Format, and Shared Resource Manager hierarchical directories. All these directory organizations are available through normal Pascal file operations. Old files (generated under Pascal 1.0) are all still fully useable. However, the new system can generate files and discs which cannot be properly interpreted by the old file system. In particular the 1.0 system could generate and process directories having either 77 or 233 entries; 2.0 can generate directories with an arbitrary number of entries.

The names of system files have been changed. This was necessary because they were longer than allowed for the LIF directory format. We wanted customers to be able to run completely under LIF if they wish, for compatibility with BASIC. The name changes are:

Pascal 1.0 file names		Pascal 2.0 file names
SYSTEM.LINKER	==>	LIBRARIAN
SYSTEM.EDITOR	==>	EDITOR
SYSTEM.FILER	==>	FILER
SYSTEM.COMPILER	==>	COMPILER
SYSTEM.ASSMBLER	==>	ASSEMBLER
SYSTEM.LIBRARY	==>	LIBRARY
SYSTEM.TABLE	==>	TABLE
SYSTEM.INITLIB	==>	INITLIB
SYSTEM.MISCINFO	==>	MISCINFO
SYSTEM.STARTUP	==>	STARTUP

Object Code Incompatibility

Note

Because of the many file system changes, object code compiled under Pascal 1.0 will **not run** under release 2.0, and in fact can't even be loaded.

It is also incompatible in the other direction. This means programs need to be recompiled to be run under Pascal 2.0. The requirement of recompilation was unavoidable in order to cleanly support remote file access (Shared Resource Manager).

If Pascal 2.0 is configured to use the Workstation 1.0 directory organization as the primary directory type, recompilation should be all that's necessary. If you want to use LIF directories, in some cases file names used by a program may have to be shortened.

IO Drivers

Some reorganization has occurred in the drivers themselves, necessitated by the new concept of Transfer Method. In release 1.0, some drivers were not properly part of the IO Library -- there existed some short-circuit paths. This has been corrected, and the structure of all drivers regularized. Also, new drivers have been added to support new peripherals.

New Peripheral Support

The following peripherals are supported by Pascal 2.0.

- The CS-80 discs (7908 family) are supported, including the streaming backup tape drive.
- The Shared Resource Manager is fully supported.
- The 8920x and 9121 flexible disc drives are supported.
- Several new versions of the 9134 micro-Winchester disc are supported. They look like one big medium instead of four smaller ones.

Certain more obscure features are supported, too. For instance, the 2.0 system can be fairly easily tailored by the customer to run from a terminal instead of the built-in CRT and keyboard.

Miscellaneous

Supervisor vs. User State

Pascal 1.0 ran all programs in the 68000's "supervisor mode". User programs now run in "user mode", using the USP (User Stack Pointer). Interrupts run in "supervisor mode", using the SSP. This would affect programs which were written to call routines in the Boot ROM. Since the Boot ROM entry points were not documented in the 1.0 system, few if any customer programs will be affected.

Global Space

Up to 65k bytes of global space is now available. This change involved a redefinition of the use of register A5, which now points to an address 32k bytes BELOW the start of globals rather than above the first global variable. Consequently, routines in the Boot ROM cannot any longer be called directly; a small interfacing routine is now required to set up the registers and fool the TRY-RECOVER mechanism. If you don't understand that, it isn't important at this time.

Software Tools Used for System Generation

Before wading into the deep water, some mention should be made of the software tools used to generate Pascal 2.0.

Assembler and Librarian

The Assembler and Librarian supplied with your system are the same ones we ourselves used to generate the system. At the end of the section of this document describing the Boot ROMs is an example using the Assembler and Librarian to create a bootable disc.

Pascal Compiler

The Compiler supplied with your system is **not** the same one we used; but we believe it will be able to do everything you will need to do.

The Compiler we used differs from the one you received in that it supports a few language extensions which are enabled by the directive `$MODCAL$`. "Modcal" is the name of a system programming language used within Hewlett-Packard.

Most of the Modcal features can be enabled in your Compiler by the directive `$SYSPROG$`. These system programming features are described in the *Pascal Language System User's Manual* Compiler chapter.

The remaining Modcal features (not enabled by `$SYSPROG$`) are used rarely or not at all in the Pascal 2.0 system. We don't want to use them, if it can be avoided, because some of these features are experimental or architecture-dependent and may not survive the tests of time, acceptance and standardization.

In addition to information about `$SYSPROG$` extensions, the *Pascal Language System User's Manual* also discusses how Pascal uses the stack for parameter passing and access to non-local variables. This is useful information if you want to call an assembly language routine. The User's Manual chapter on the Assembler has examples.

Memory Allocation of Variables

In a system programming context, sometimes it is useful to know how the Compiler will allocate space for variables in memory. The 68000 processor is sensitive to the address alignment of variables in some cases. Here are the rules the Compiler follows in laying out variables.

- **Arrays:** Alignment is always 2 (that is, arrays are aligned to even addresses -- word boundaries).
- **Records:** A record which is part of a packed structure is itself not packable; within the containing structure, the record's alignment will be 1 (any byte boundary) if the entire record fits in a single byte; otherwise alignment will be 2. If the fields of the record are themselves packed, its alignment will be 2.
- **Sets:** Sets are not packable. Alignment is 2.
- **Pointers:** Not packable, alignment is 2.
- **Chars:** Packable, alignment is 1. Packed size is 8 bits.
- **Booleans:** Packable, alignment is 1. Packed size is 1 bit.
- **Enumerated scalars needing less than 17 bits:** Packable, alignment is 2. Unpacked size is 16 bits, packed size is number of bits needed.
- **Scalars needing 17 or more bits:** Packable, alignment is 2. The packed field must be accessible in one long (32 bit) move. This may force the packed field to be aligned on an even byte boundary. Unpacked size is 4 bytes.
- **Integers:** Integers get 32 bits and are not packable. Alignment is 2.

The Compiler directive `$TABLES$` causes the Compiler to print out a description of the space allocated for types and variables in a program. Use it if in doubt.

When writing system code, it is usually perfectly reasonable to enable `$DEBUG$` and use the Debugger to step through your stuff. However, this won't work well for interrupt routines! Likewise `$RANGE ON$` and `$STACKCHECK ON$` are generally reasonable during debugging. In fact, it may be undesirable to ever disable stack overflow checks.

You will almost surely want to specify `$IOCHECK OFF$` in system code.

Chapter 2

Peripheral Configuration

Introduction

When Pascal 2.0 boots up, it determines what peripheral devices (local and remote mass storage devices, printers) are connected to the computer, and makes them accessible to the File System. This service is called "configuration", and is performed by a program called TABLE which executes while the system is booting.

Pascal 2.0 is self-configuring, which means no special action is required for the system to recognize most peripherals. In other words, this section explains a subject which most users need not worry about.

Note

Pascal 1.0 users **should** read this section because Pascal 2.0 will, unless told otherwise, pick the wrong disc directory organization for your existing discs. This is discussed later under the subheading: *Alternate Directory Access Methods*.

Overriding the automatically produced configuration, although not normally necessary or desirable, is explained in the next section.

Definitions

- **Peripheral:** an IO device such as a printer or disc. Devices such as plotters and digitizing tablets are also peripherals, but they are accessed through the IO library rather than the Pascal file system. For the present discussion we use the term to refer only to devices accessible through File System operations.
- **Interface:** the electronic circuitry which connects the computer's high-speed internal bus to lower speed physical peripheral devices. Interfaces are either built-in, like the standard HP-IB (IEEE-488) port at the back of your computer, or plug into the IO backplane. Most of the peripherals supported by the Series 200 Computers are designed to connect through an HP-IB interface.
- **Select code:** a number between 0 and 31, the "address" or name by which an interface is identified and referenced. When a peripheral operation is performed, it takes place through an interface which is said to be "on a select code". Most interface cards which plug into the IO backplane have switches which can be set to indicate the select code to which the interface will respond. The built-in interfaces have fixed select codes.

- **Bus address:** when several peripherals are connected to the same HP-IB interface, a bus address is required (in addition to the select code) to designate the particular peripheral referenced by an IO transaction.
- **Device specifier:** by convention in the Pascal, BASIC and HPL systems, when select code and bus address are used together to address a peripheral, they are concatenated into a single number. Thus the device at address 1 on select code seven is referenced as "701", which is derived by multiplying the select code by 100 and adding the address. NOTE: some HP products contain, within a single package, several peripheral devices which must be addressed separately.
- **Boot device:** the peripheral where the Boot ROM found and loaded the Pascal operating system. The newer Boot ROMs have a complicated search pattern which allows booting from just about any drive in any HP mass storage product, including the Shared Resource Manager.
- **Volume:** a volume is a named mass storage medium or a named, unblocked file device such as a printer. The name of a mass storage volume is found in its directory; the name of an unblocked device is found in its Unit Table entry. There may be several volumes on one physical storage medium. The volume may be mounted (in a disc drive) or not.
- **Unit Table:** the Pascal system provides for up to 50 units, designated #1 through #50. They are represented by a 50-entry array called the Unit Table or "Unitable". Each entry fully specifies the association of one logical unit to a physical peripheral, with such information as the device specifier and driver procedures to be used for IO operations to the unit.
- **Unit:** an entry in the Unit Table.
- **System volume or System unit:** the Pascal system distinguishes one mass storage unit to be used for special purposes. This "system volume" is where the date and any AUTOSTART file are found at boot time, where the system looks first for special files such as the Compiler and Editor, where Workfiles are stored, and where an intermediate file is stored during interpretation of a Stream (command) file.
- **Directory Access Method or DAM:** each mass storage unit has a directory describing the files it contains, the type of each file and so forth. Many different directory organizations are used within HP, and data on a disc can't be interpreted properly unless it is accessed using the correct Directory Access Method. Pascal 2.0 supports three DAMs: the "Workstation" format compatible with Pascal 1.0 systems; HP's "Logical Interchange Format" or LIF directory; and the Shared Resource Manager's hierarchical directory.

The purpose of the Configuration Process is to fill in entries of the Unit Table so it correctly associates logical units with peripheral devices and the software required to drive those devices.

Principles of Auto-Configuration

The approach to auto-configuration is straightforward. First, interface select codes 7, 8, and 14 are scanned; on each select code, if an HP-IB interface is present then bus addresses 0 to 7 are interrogated for peripheral devices. (Select code 7 is the built-in HP-IB port. Most HP peripherals identify themselves when asked politely.)

In the Unit Table, certain unit numbers are preferentially auto-configured to particular devices. If an interesting peripheral is found, it is assigned to its preferred unit number. Hard discs are a special case. Auto-configuration assumes at most one hard disc will be present, and if a hard disc is found, it is partitioned into several volumes which are assigned consecutive unit numbers beginning with #11. The size of each volume and the number of volumes depends on the type of disc.

Even though the Pascal workstation by default partitions its hard discs into multiple logical volumes, currently no other mainframes or operating systems do. Because of this there is an exception to the automatic partitioning: if the first directory exists, but one or more of the remaining directories do not, then auto-configuration will treat the disc as a single volume. This allows direct interchange of data with other systems via hard discs, notably, interchange with HP BASIC. BASIC always initializes discs so that there is a single volume on the medium.

Auto-configuration observes the rule that the device from which the system booted should be accessible, even if its preferred unit assignment would have otherwise been overridden by a higher-priority preference for some other peripheral.

Finally -- the System Unit is chosen according to the rule that it should be the unit from which Pascal booted, **unless** the boot device is some variety of 5.25-inch or 3.5-inch medium; in that case, it is selected from among the mass storage units by a scheme which prefers bigger, faster discs if they are present. The precise pecking order is: hard disc, 9885 drive 0, 9895 drive 0, Shared Resource Manager, 5.25-inch built-in disc drive 1, 5.25-inch built-in disc drive 0.

Note

The "left" and "right" drives: in the 9826 and 9836 mainframes, the right-hand or only drive is drive zero; the other is drive one. But for peripherals with two identical drives in an external package, the left-hand drive is drive zero! This applies to the 8290x series, the 9895, and the 9121.

The resulting configuration is covered by the following table:

Unit	Nominal assignment
1	CONSOLE: the CRT (built-in display)
2	SYSTEM: the keyboard
3	right-hand minifloppy (drive 0) or 8290x minifloppy (drive 0) or 9121 microfloppy (drive 0)
4	left-hand minifloppy (drive 1) or 8290x minifloppy (drive 1) or 9121 microfloppy (drive 1)
5	Shared Resource Manager (remote mass storage)
6	PRINTER: the system printer
7	9895 floppy disc (drive 0)
8	9895 floppy disc (drive 1)
9	9885 floppy disc (drive 0)
10	9885 floppy disc (drive 1)
11-40	hard discs
41	tape backup device in CS/80 (7908-family) discs
43	same device as unit #3, but alternate DAM
44	same device as unit #4, but alternate DAM
45	SRM system volume if appropriate
47	same device as unit #7, but alternate DAM
48	same device as unit #8, but alternate DAM
49	same device as unit #9, but alternate DAM
50	same device as unit #10, but alternate DAM

Units #1 through #6 are assignments which are not intended to be changed, regardless of the rest of the unit assignments. Even if you ultimately generate a non-standard configuration, you ought to leave these assignments alone. (Actually even these units can be reassigned, but lots of programs are written to depend on the standard assignments.)

The purpose of units #43, #44, #47, #48, #49 and #50 is explained in the next section. The hard disc units are discussed in more detail later.

Alternate Directory Access Methods

The files on a disc are found and accessed by means of a directory which describes where the files are located, how big they are, what types of data they contain, and so forth. The directory is of course stored on the disc itself. There are many reasonable ways to organize discs, depending on one's purposes; these alternative organizations are called "Directory Access Methods", or DAMs. A disc can be read or written by the File System only if the correct DAM is used.

Pascal 2.0 supports three disc organizations: the Pascal 1.0 "Workstation" format, HP's Logical Interchange Format (LIF) and the Shared Resource Manager's hierarchical directory structure. The DAM used for each logical IO unit is selected by the TABLE configuration program executed at boot time. (To be more precise, the SRM DAM is supported in the SRM itself; what Pascal supports is the communication of DAM requests to the SRM.)

Of these three DAMs, the SRM method can only be used with remote mass storage over an SRM hookup. The other two methods can be used with any local mass storage device. Auto-configuring selects one as the "primary" DAM, the other as "secondary". (Sometimes the word "alternate" is used instead of "secondary".) The primary DAM is the one used for units #1 through #40 (except #5, auto-configured as the SRM unit). The secondary DAM, used by units #43, #47, #48, #49 and #50, is available to allow discs in the secondary format to be used by Pascal programs and the Filer utility.

The secondary DAM is in no way restricted from normal use by the File System; discs in the secondary DAM units are directly readable and writeable by Pascal programs.

If Pascal 2.0 is booted up just as shipped, the primary DAM will be LIF. In this case, Pascal 1.0 discs must be accessed through the alternate DAM units. To make the Pascal 1.0 format the primary DAM, do the following:

1. Boot up the original system as supplied.
2. Use the Filer's "F" (Filecopy) command to copy SYSVOL:CTABLE1.0 to BOOT:TABLE
3. Turn power off and reboot the computer.

To return to using LIF as the primary DAM, use the Filer to copy CONFIG:CTABLE2.0 to BOOT:TABLE.

Selecting the Primary DAM

If you are a new user and have no existing discs in the Workstation directory format, we recommend that you use the system as supplied, with LIF the primary DAM. LIF is an HP corporate standard for information interchange among computer systems.

If you have discs generated by Pascal 1.0 you may change the primary DAM as just described, or access them through the limited number of alternate DAM units, or transfer your files to new LIF volumes. The choice is primarily one of convenience, although in the long run there may be some advantages to LIF. Since Pascal programs which ran under the 1.0 release must be recompiled to run under Pascal 2.0, you may choose to convert your discs as well.

The boot disc must have a LIF directory unless the boot device is a Shared Resource Manager.

Comparison of LIF and Workstation DAMs

In both DAMs, all the space allocated to a single file is contiguous. Consequently, if the free disc space is fragmented, either DAM may be unable to create a new file of a specified size even though there is enough total free space on the disc.

In either DAM, it may not be possible to extend (append to) an existing file even if the disc volume has some free space. A file in either DAM can only be extended if there happens to be free space immediately following the file. Appending to files was not allowed at all in the Pascal 1.0 release.

The case of letters is significant in LIF file identifiers; the file called 'Charlie' is not the same as 'charlie'. Case is not significant under the Workstation DAM (more precisely, file names are automatically converted to upper case in Workstation disc directories). The same comments apply to volume names in the two DAMs.

Workstation DAM file names may be up to 15 characters long. LIF names are restricted to 10 characters. In many cases this difference need not be a problem. Most file names used by the Pascal system end in a five-character suffix such as '.CODE' ; hence the useful part of such names is 10 or fewer characters. The LIF DAM implementation encodes recognized standard suffixes into the "tail" of a LIF file name, so that 9 characters are available for the significant part of the name. This encoding is transparent, as is the decoding back into the full suffix when necessary. It is done using the programming technique called "mirrors".

File Interchange between WS1.0 and LIF

Exchanging files between Workstation 1.0 volumes and LIF volumes is possible.

The general process is as follows:

1. Invoke the Filer by typing: F while at the Command prompt level.
2. Put the source disc in a drive configured for its type of DAM, and the destination disc in a drive configured for its DAM.
3. Use the Filer's F (Filecopy) command to move files from one disc to the other. The Filer commands will work with either DAM.

Note that the alternate-DAM units allow either DAM to be used in the same drive. For instance, you can transfer a file from #43 to #3, both of which are assigned to the right-hand minifloppy drive in a 9836 or 9826. Example Filer command:

```
F #43:CHARLIE.TEXT,#3:$
```

The Filer will tell you to when to swap discs.

Remember: the name of a file in a Workstation disc may be too long for LIF; you may have to invent a shorter name.

By the way, it's a good idea to develop the habit of using uppercase letters in the names of LIF files. LIF file name suffixes must be uppercase!

Note

Directories created by the Pascal 1.0 Workstation have either 77 or 233 entries, whereas WS1.0 directories created by Pascal 2.0 have a variable number of entries specified by the user. Thus you can use Pascal 2.0 to create WS1.0-format discs which aren't readable by the Pascal 1.0 system, whereas all 1.0 directories are readable by 2.0.

Supported Mass Storage Products

There are two general types of discs supported by Pascal 2.0; "flexible" discs (floppy discs) and "hard" discs (fixed discs).

The flexible disc drive products supported by Pascal 2.0 are:

- Built-in minifloppy drives (9826 and 9836)
- 8290x 5.25" minifloppy drives
- 9121 3.5" microfloppy drive
- 9885 8" single-sided disc
- 9895 8" single or double-sided disc

The following table shows which hard disc products can be used with the Pascal 2.0 system. The table also shows that more than one logical unit will be allocated to each disc when the system auto-configures itself. In these cases the disc has been partitioned into several non-overlapping volumes, and the table indicates the size of each volume in sectors. A sector is 256 bytes.

If an application requires larger volumes, it will be necessary to override the automatic configuration parameters as described later in this chapter.

product identifier	volumes assigned	space per volume in sectors	
-----	-----	-----	
9134A/9135A	11-14	4500	(looks like 4 drives)
9134A/9135A (opt. 10)	11-14	4712	(looks like 1 drive w/4 vols)
9134B/9135B	11-19	4185	except #19 = 4340 sectors
9134C/9135C	11-24	4052	except #24 = 4054 sectors
7908	11-26	4025	except #26 = 4375 sectors
7911	11-37	4032	except #37 = 4992 sectors
7912	11-40	8512	except #40 = 9408 sectors

BASIC and Pascal File Interchange

You may wish to exchange data between the BASIC and Pascal environments. There are a few rules you should follow.

1. Pascal and BASIC treat LIF directories on flexible discs similarly. ASCII text files are intended to be read by both systems.
2. It was mentioned earlier that Pascal compresses the suffix of user file names in order to effectively allow longer file names. BASIC doesn't know about compressed names, so the BASIC program needs to invert the compression algorithm. This is very easy, and is described in the chapter, "Programming with Files". Essentially, Pascal chops off the dot and the suffix, then appends the first letter of the suffix and enough trailing underscores ("___") to make a 10-character name. Thus XXX.TEXT becomes XXXT_____, which is the name BASIC will see.
3. BASIC can't deal with more than one LIF directory on a disc medium; Pascal, unless told otherwise, wants to divide large discs into several volumes each with its own directory.

If a disc is initialized by BASIC, Pascal and BASIC will both see the disc as one very large volume. Pascal's preference to partition the disc is overridden by what BASIC actually did.

If a disc is initialized by Pascal and partitioned into multiple volumes, BASIC will only see the first volume and will not be able to access any part of the disc beyond the first volume. Pascal will see all the volumes.

See the next chapter for information on forcing Pascal to treat a partitionable disc as a single volume.

Library Management

A **library** is a file produced by a language processor such as a Compiler or Assembler, or by the Librarian. Libraries contain zero or more modules of machine code ready to link or relocate. The concept of a code module is discussed extensively in the *Pascal Language System User's Manual* in the chapters which describe the Compiler and Librarian.

This presentation assumes you are familiar with that material, and discusses management of two system libraries of particular importance, called **LIBRARY** and **INITLIB**.

When a code file is loaded into memory (to be run or to be kept resident via the **P** command), it usually contains "unsatisfied references" to code and data objects not included in the file itself. The linking loader follows a predetermined search pattern in looking for these objects. First it looks in the modules currently resident in memory, most recently loaded modules first. Then it looks in the system **LIBRARY** file.

If any objects a program needs cannot be found by this search, the missing items will be listed on the CRT with a message that "The above external references were unsatisfied."

LIBRARY usually resides on your system volume, although the Command Interpreter's "What" command can be used to specify otherwise. The one provided on your **SYVOL** disc is a very minimal library; additional modules for Graphics and Device IO are supplied on the **LIB** disc, and should be added to your system **LIBRARY** before you try to compile or run most Graphics or IO-oriented applications. This section tells how to add stuff to your system **LIBRARY**.

The **INITLIB** file contains modules which are automatically made resident when the computer is "booted up", primarily software required to run the File System and peripheral devices such as printers and disc drives; the system **LIBRARY** file contains modules which are used more occasionally and therefore are loaded on demand.

INITLIB lives on the **BOOT** disc (where else?). As supplied with your system, it contains software support for the most commonly used peripheral devices. The disc called **CONFIG** contains several more libraries which you must add to your **INITLIB** if you plan to use certain less common peripherals. This section also tells you how to add stuff to **INITLIB**.

The System LIBRARY

As supplied on your original discs, the system LIBRARY contains just four modules:

RND	Random number generator.
HPM	Heap management (new/dispose).
UIO	Unit IO (UCSD Pascal compatible low-level IO).
LOCKMODULE	Interlock code for multiple access to shared files.

("UCSD Pascal" is a trademark of the Regents of the University of California.)

The LIB disc contains three more library files with modules you can add to LIBRARY if they are needed. Almost all the procedures in these modules are described in the *Pascal Procedure Library User's Manual*. They are not in the standard LIBRARY so the SYSVOL disc doesn't get unnecessarily crowded, and also to simplify future enhancements. These files are called INTERFACE, IO and GRAPHICS.

The INTERFACE File

The modules in this file contain no object code at all, only interface specifications for the Compiler to read when the modules are IMPORTed. The actual object code is always found in memory by the linking loader; it is part of the Operating System kernel or INITLIB. To use these modules successfully you need information found later in this manual.

For reference, the modules in INTERFACE are:

ASM	KBD
SYSGLOBALS	INITKBD
MINI	KEYS
BOOTDAMMODULE	KEYSINIT
LOADER	CRT
INITLOAD	INITCRT
ISR	BAT
MISC	INITBAT
FS	CLOCK
INITUNITS	INITCLOCK
LDR	CI
SETUPSYS	CMD

The IO File

This file contains almost the entire Pascal Device IO library; the only parts not in this file are always resident, loaded from INITLIB. The modules of the IO library which are always resident and therefore not in this file are:

```
IODECLARATIONS
IOCOMASM
IOLIBRARY_KERNE
GENERAL_0
```

The modules in IO are:

```
GENERAL_1          HPIB_0          SERIAL_0
GENERAL_2          HPIB_1          SERIAL_3
GENERAL_3          HPIB_2
GENERAL_4          HPIB_3
```

The *Pascal Procedure Library User's Manual* is the reference document for the IO library, and details what modules are needed in order to provide various IO capabilities. Here are some useful hints:

- GENERAL_1, GENERAL_2, GENERAL_3 and HPIB_1 cover most of the commonly used IO facilities. These modules are also required by the Graphics library.
- GENERAL_4 adds the "Transfer" capability (buffered, overlapped transfers of blocks of data).
- HPIB_0, HPIB_2 and HPIB_4 provide various levels of HP-IB (IEEE-488) capability.
- SERIAL_0, SERIAL_3 provide IO through the 98628 "smart" data communications card, the 98626 "dumb" RS-232 card and the "DATA COMMUNICATIONS" port on the back of the 9816 mainframe.

The GRAPHICS File

This file is the Device-independent Graphics Library, or "DGL". The capabilities provided are a compatible subset of the capabilities found in the HP-1000 series minicomputer DGL implementation. DGL is also described in the *Pascal Procedure Library User's Manual*.

The modules in GRAPHICS are:

```
DGL_TYPES
DGL_VAR
DGL_ARAS
DGL_RAS
DGL_MAIN
DGL_LIB
DLG_INQ
```

Adding Modules to the System LIBRARY

Adding files to LIBRARY is a straightforward exercise in using the Librarian -- see the relevant chapter of the *Pascal 2.0 Language System User's Manual* if you don't know how. Actually, one never "adds" files to a library; instead, a new library is created which includes copies of modules from other libraries.

Specify LIBRARY. as your first input file. Be sure to type the dot after the name, since this name doesn't end in the usual .CODE suffix. Copy all the original modules using the A command. Then insert the LIB disc in a drive and specify whichever additional library you want as the new input file (for instance, LIB:IO.). Again, be sure to include a dot after the name of any library file that doesn't have the .CODE suffix.

You may transfer particular modules, or all of them. You can add modules from as many files as necessary. The order of modules in LIBRARY is unimportant.

When you have all the modules you want, keep the output file with the K command then Quit the Librarian.

Then use the Filer to delete the current LIBRARY file, and rename the one you created. For instance, if you specified that the new library being created was to be called MYLIB, the resulting file would be MYLIB.CODE; with the Filer's F (Filecopy) command, copy MYLIB.CODE to LIBRARY on your system volume.

(Note: if you had specified to the Librarian that the new file name should be MYLIB. with a period after the name, the appending of the .CODE suffix would have been suppressed. Thus in fact you could have specified that the new output file be called LIBRARY and avoided the file copy step. But that is a risky practice; you should be sure you have what you want before destroying the old library.)

Notes and Possible Problems

The disc to which the output file is being written must **never** be removed during the process of making the new library. If your computer doesn't have at least two mass storage volumes available, you will have to create a memory-resident volume using the Command Interpreter's M command and temporarily hold the output file there in memory.

The amount of memory that must be available to make a new LIBRARY using a single disc and a memory-resident volume depends on how much stuff will be put in the new library; but if you were to add both IO and GRAPHICS to the original LIBRARY you would need to allocate about 300 blocks of 512 bytes to the memory volume (153,600 bytes).

If you add enough modules to the output file (beyond those in IO and GRAPHICS), the Librarian may eventually report the error, "file header full". If this happens to you, start over and use the H command to specify a larger library header before specifying the output file. A header specification of 58 is usually big enough for most situations.

Even if you are using the INTERFACE file, there is probably still no point in adding it to the system LIBRARY. The reason is that, since it contains no actual code, it is only used at compile time. You can use the Compiler's \$SEARCH directive to tell the Compiler where to look for modules in the INTERFACE library.

The section, *Setting Up the Shared Resource Manager* gives a very detailed cookbook example of using the Librarian with a memory-resident volume. You might wish to look at that if you run into problems.

INITLIB and Memory-Resident Modules

As mentioned previously, the INITLIB supplied on your original BOOT disc contains a reasonably complete set of peripheral driver software. You may wish to install other drivers, which are supplied on the CONFIG disc; or to conserve memory you may wish to remove items you don't need.

Unlike the system LIBRARY, modules in INITLIB are somewhat order sensitive -- certain modules, if present, must precede others in INITLIB. The list which follows shows all the modules supplied with Pascal 2.0. If you add or delete INITLIB modules, all the modules which are present in the resulting INITLIB should appear in the order listed.

The "Recommendation" column advises you about the importance of each module. Items marked "Required" are essentially required in INITLIB. It is sometimes possible to remove these items, but the system is likely to be crippled.

The notation "Almost" means "almost required" -- we don't recommend removing these unless you have determined for sure they aren't needed, because they are part of the normal functioning of the system.

Items marked "Development" are usually needed in a software development environment but may not be required for a particular application. Items marked "Optional" are optional unless required by a particular system configuration.

Module	Where found	Recommendation
-----	-----	-----
KERNEL	BOOT:INITLIB	Required
KBD	BOOT:INITLIB	Required
KEYS	BOOT:INITLIB	Required
CRT	BOOT:INITLIB	Required
BAT	BOOT:INITLIB	Required
CLOCK	BOOT:INITLIB	Required
DEBUGGER	BOOT:INITLIB	Optional
PRINTER	BOOT:INITLIB	Development
DISCHPIB	BOOT:INITLIB	Development
AMIGO	BOOT:INITLIB	Optional
IODECLARATIONS	BOOT:INITLIB	Required
HPIB	BOOT:INITLIB	Almost
DMA	BOOT:INITLIB	Development
REALS	BOOT:INITLIB	Development
ASC_AM	BOOT:INITLIB	Development
WS1_0_DAM	BOOT:INITLIB	Development
TEXT_AM	BOOT:INITLIB	Almost
CONVERT_TEXT	BOOT:INITLIB	Almost
LIF_DAM	BOOT:INITLIB	Almost
CS80	CONFIG:CS80	Optional
DISC_INTF	CONFIG:DISC_INTF	Optional
DATA_COMM	CONFIG:DATA_COMM	Optional
GPIO	CONFIG:GPIO	Optional
RS232	CONFIG:RS232	Optional
SRM	CONFIG:SRM	Optional
F9885	CONFIG:F9885	Optional
LAST	BOOT:LAST	Required

If you try to make a Boot disc with an INITLIB containing everything in this list, it won't all fit on a 5.25-inch flexible disc! However, very few applications really require all these drivers. For instance, it is unlikely you will need the F9885 module, since the 9885 is an obsolete disc which is almost never used with the Series 200 family. Likewise it is unusual to need both RS-232 and DATA_COMM, which drive two different serial interface cards, or AMIGO and CS/80 which drive hard discs from two different families.

In the unlikely event that all the stuff you really need won't fit on your Boot disc, there are still two ways to make it available.

If your computer has a Revision 3.0 or later Boot ROM (but **not** the one called 3.0L, which is used with low-cost versions of the 9816 computer), it is capable of booting from external mass storage devices, so you can make a "total" INITLIB, store it on a larger disc, and should have no problems. If your computer has an earlier Boot ROM or no external mass storage, you can put the modules you need (which are **not** labeled "optional" in the list above) on your boot disc and use the Librarian to put the optional modules together in a secondary library on another disc. Then after booting up, simply EXECUTE the secondary library as if it were a program. The additional modules will install themselves.

A very detailed example of how to add modules to INITLIB is provided later. It shows all the steps required to install Shared Resource Management software in your Workstation and set up the central SRM.

Here is information on what additional modules (beyond those in the originally supplied INITLIB) you must install if your workstation uses various peripheral devices or interface cards.

98620 Direct Memory Access Interface

The driver for this interface is module DMA, which is present in the original INITLIB. The interface is used in conjunction with other cards.

98622 GPIO (16-bit parallel) Interface

To drive this interface, add module GPIO.

98625 High-speed Disc Interface

This is a form of HP-IB interface, but it is only for use with discs and, oddly enough, printers. Add module DISC_INTF to drive this card. Also requires module DMA (already present in supplied INITLIB). See the comments below about module DISC_INTF.

98626 Serial RS-232 Interface

To drive this interface, install module RS232.

98628 Data Communication Interface

To drive this interface, install module DATA_COMM.

98629 Shared Resource Management Interface

Very detailed instructions on how to install the SRM software are given later as an example. To use the SRM, add modules DATA_COMM and SRM.

External Disc Drives of the Command Set 80 Family

These are disc models 7908, 7911, 7912.

Add module CS80. Note that DISCHPIB must also be present. If your machine has a direct-memory access interface (98620), module DMA must be present to use it. This is not required for CS/80 discs but improves performance somewhat.

External Disc Drives of the Amigo Family

These are models 8290x (5.25" flexible disc), 912x (3.5" micro-Floppy), 913x (Winchester-type fixed disc) and 9895 (8" flexible disc).

The required modules are AMIGO and DISCHPIB. Both of these are already present in the supplied INITLIB. In addition, performance of the 913x series discs is substantially improved by a 98620 DMA interface and module DMA (already present).

9885 8-inch Flexible Disc Drive

This disc requires you to install module F9885. Makes use of module DMA (already in original INITLIB), as well as the 98620 DMA interface and 98622 16-bit parallel interface. Note that module GPIO is not required even though it is the module which "normally" drives the 98622 card.

Printers

Module PRINTER (already present in supplied INITLIB) is required to drive all printers, regardless of the type of interface being used. Additionally module HPIB (already present) is required for HP-IB printers.

Serial printers can be used with the 98626 Serial Interface card if module RS232 is added to INITLIB and the TABLE configuration program is modified to specify the select code of the serial interface.

Module DATA_COMM can be used with the 98628 Data Communication interface to drive a serial printer; again the printer select code specified in the TABLE configuration program must be altered. You may also need to modify the value of local_printer_timeout in CTABLE, because many serial printers have a large internal buffer which gets filled and takes a long time to be dumped onto the paper. See the description of how to modify CTABLE.

Graphics Devices

To talk to HP plotters, via HP-IB, requires module HPIB (already present in the supplied INITLIB). In addition, the Graphics library requires other modules to be in the system LIBRARY rather than INITLIB -- see the discussion of the system LIBRARY above.

Notes on Various Modules

- **KERNEL** is the "core" of the system, containing the Linking Loader and basic File System support. It needs to be there.
- **KBD, KEYS, CRT, BAT** and **CLOCK** are responsible for the keyboard, foreign character set, display, battery backup and clock. They are broken out into several small modules so they may be replaced individually if desired. They (or some code with equivalent function) must be present.
- **DEBUGGER** is the interactive debugging tool. It is usually resident, but is a good candidate for removal from completed applications because it is a particularly dangerous thing to put in the hands of non-programmers.
- **PRINTER** is required to drive all printers, regardless of the type of interface electronics being used. It supports serial as well as **HP-IB** printers; a serial interface driver must be present in **INITLIB**, and the **TABLE** configuration program must be adjusted to select that driver. To use serial printers, you may have to modify `local_printer_timeout` in the **CTABLE** program. See the material on modifying **CTABLE**.
- **DISCHPIB, AMIGO, CS80** are all related. To use any external disc drive connected via **HP-IB** you must have:

DISCHPIB and **AMIGO** for disc models:

9895	(8" flexible disc)
912x	(3.5" micro-Floppy disc)
913x	("Winchester" type fixed disc)
8290x	(5.25" flexible disc)

DISCHPIB and **CS80** for fixed discs of the Command-Set 80 series, models: 7908, 7911, and 7912.

- **DISC_INTF** is required to use the 98625 high-speed disc interface card. **DMA** is also required for this card. The 98625 is primarily used for achieving maximum performance from the **CS/80** discs. The performance improvement with these discs is negligible for most purposes, and only shows up in the transfer of very large blocks of data. It might be useful in data logging applications. In addition the 98625 can be used for **HP-IB** printers, although it seems a silly thing to do, and it can be used with the 9895, 9121, 9133 and 9134 discs. With these discs there will be no performance improvement over the standard **HP-IB** interface with **DMA**.

Do not use the 98625 with the 8290x or 9135 discs; it won't work!

- **IODECLARATIONS** is the lowest level of device **IO** support. Although it is possible to construct loadable systems without this module, no normal File System **IO** would work.
- **HPIB** is the lowest level support for the Hewlett-Packard Interface Bus, which is **HP's** implementation of the **IEEE-488** Standard. Most **HP** peripherals use **HPIB**, so there isn't much point in ever removing this module.
- **DMA** is the module which runs the 98620 Direct Memory Access interface card. **DMA** provides very high speed data transfers. It is required for the 98625 interface card and highly recommended with the **CS/80** and **913x** families of discs.

- REALS is the floating-point mathematics support package.
- ASC_AM is the Access Method responsible for blocking and unblocking text files with the LIF-ASCII structure. LIF stands for Logical Interchange Format, a common file interchange structure supported by most HP computers. Since this is one of the formats used by the BASIC language system, it is a good thing to have around. It is also the format used by the SRM for spooled printer files.
- WS1.0_DAM is the Directory Access Method used by the Pascal 1.0 system, a predecessor to the one you are using. This module lets the system read and write discs in that format. This DAM can be removed if you have no need to read or write discs compatible with the Pascal 1.0 system.
- TEXT_AM is the Access Method used to block and unblock text files created with the .TEXT suffix. These are the files normally created by the Editor (unless the user specifies otherwise). The .TEXT file structure is compatible with text files generated by UCSD Pascal systems.
- CONVERT_TEXT is a module used by the Compiler and other subsystems to convert among the various representations of text files. It should be present in INITLIB.
- LIF_DAM is the Directory Access Method required to read and write HP Logical Interchange Format disc directories. LIF is the primary directory organization used with Pascal 2.0, and so this module is normally present. If you configure your system to use WS1.0 as the primary directory method, you may remove LIF_DAM.
- DATA_COMM is the module required to drive 98628 and 98629 interface cards.
- GPIO is the module required to drive the 98622 16-bit parallel interface card.
- RS232 is the module required to drive the 98626 serial RS-232 interface card, and the built-in serial interface in a model 9816 computer.
- SRM is required to drive the 98629 Shared Resource Management interface. Module DATA_COMM is also required.
- F9885 is required for model 9885 flexible disc drives. These discs also require module DMA, and the 98620 and 98622 interface cards. Module GPIO is **not** required for the 9885 even though a GPIO card is used.
- LAST is required in every case, and **must** be the last module in INITLIB. The purpose of this module is to actually start the system running after the contents of INITLIB have been loaded and installed in memory. LAST principally does two things: load and execute the IO configuration program TABLE; and load and execute a program called STARTUP, which is usually the Command Interpreter but may be a user program.

Chapter 3

Modifying the Configuration

Introduction

The work of configuring the system to recognize peripheral devices is performed by a program called `TABLE`, which is executed during the boot-up process. The information in this section explains how `TABLE` works, and how to modify it to achieve special effects. See Chapter 4 for details on configuring SRM systems.

General process

The Pascal source of `TABLE` is provided on the `SYSVOL` disc distributed with every copy of the system, called `CTABLE.TEXT`. Once you have decided how you wish to modify `CTABLE`, make your modifications to a copy. Compile this modified program, yielding an object code file (e.g. `MYTABLE.CODE`).

Now you can simply execute `MYTABLE` to see if the results are correct. In fact, the system can be reconfigured any time by executing a version of `TABLE`. The program does its work by assigning new values into fields of the system data structure called the "Unitable". Another section of this document gives a detailed explanation of the meaning of the various `Unitable` entries.

When you are quite sure the new table is correct, use the `Filer` to copy the compiled code to your `BOOT` disc. The name of the copy should be `TABLE` (not `TABLE.CODE`) in order to be recognized during boot-up. Be careful here! If there is no backup copy of the `BOOT` disc with the original `TABLE`, and there is something wrong with the modified `TABLE`, you may not even be able to run your system to restore the original `TABLE`.

Depending on the size of `INITLIB`, there may not be much room on the `BOOT` disc. You may need to `Krunch` it with the `Filer` to make space. The modified `TABLE` can also be made considerably smaller by linking it to itself, which combines all the internal modules into a single module, and gets rid of module interface specifications and internal reference information. "Linking to itself" is a slightly misleading phrase, in that a new file is created with the modules internally linked into a single module.

To link MYTABLE.CODE to itself, use the following steps:

- Invoke the Librarian using the L command.
- Use the "I" command to specify the input file.
- Use the "H" command to specify a 1-block header (you specify one; the Librarian forces it to the minimum of eighteen).
- Use the "O" command to specify the output file name.
- Use the "L" command to change from Copy to Linking mode.
- Use the "D" command to remove DEF information.
- Use the "A" command to transfer all modules.
- Use the "L" command to finish linking.
- Use the "K" command to keep the output.
- Use the "Q" command to quit.

Here is the summary of commands to link MYTABLE.CODE to itself:

keystrokes	meaning
L I MYTABLE	invoke Librarian, name input file
H 1	1-block header
O TABLE	name output file
L D A L K Q	link; no DEFs; all modules; finish; keep; quit

You may need to precede the file names by the volumes where they are found. Note that the result of this operation is a file called TABLE.CODE; what you want on the boot volume is a file called TABLE with no suffix. Use the Filer to change the name after the Librarian creates the file.

Commentary on the CTABLE Program

CTABLE is a long program; for ease of study, here is a summary of its structure. You will probably want to print out the code and examine it in detail.

```
program ctable;

  module options;
    Contains declarations which MAY BE EDITED to override
    many of the system defaults.

  module ctr;          DON'T MODIFY THIS MODULE.
    Exports the table entry assignment routines, which contain
    information highly specific to HP peripheral devices.

  module brstuff;     DON'T MODIFY THIS MODULE.
    Figures out which device was the boot device.

  module scanstuff;   DON'T MODIFY THIS MODULE.
    Contains code which asks each device to identify itself.

begin

  EDIT THE MAIN PROGRAM ONLY if the desired result cannot be
  obtained by modifying the declarations in module OPTIONS.

  scan for devices on various HP-IB addresses.

  determine the nature of the boot device.

  search for "complementary" CS/80 devices.

  create temporary unit table.

  make assignments to fields of temporary table.

  optional "manual" assignments to override automatic defaults.

  copy temporary to actual system unit table.

  set prefix of Shared Resource Manager system volume.

  remove extraneous hard disc entries if necessary
  (if the hard disc was initialized to have only a single
  volume on the medium)

  assign system unit.

  set prefix of default volume on SRM.

end.
```

Modifying Module OPTIONS

This module consists only of declarations of exported types and constants. The constants are examined during execution of the main program.

```
const system_unit = 0;
```

If this constant is non-zero, it indicates which of the 50 volumes will be the system volume. When it is zero, the program makes its own choice according to the algorithm described previously.

```
type lms_dam_type = (LIF,UCSD);
const primary_lms_dam = LIF;
```

Selects the primary Directory Access Method for local mass storage devices. LIF is HP's Logical Interchange Format directory; UCSD is the format used in the Pascal 1.0 workstation release. As the name indicates, this directory structure is compatible with the organization used by the "UCSD Portable Pascal" system ("UCSD Pascal" is a trademark of the Regents of the University of California).

CTABLE expects remote mass storage devices to use the Shared Resource Manager's hierarchical directory organization.

```
type dav = record
    sc,ba,du,dv: shortint;
end;
const
    hp8290x_default_dav = dav[ sc:7, ba:0, du:0, dv:-1 ];
    hp9895_default_dav = dav[ sc:7, ba:0, du:0, dv:-1 ];
    hp9885_default_dav = dav[ sc:12, ba:-1, du:0, dv:-1];
    harddisc_default_letter = 'H';
    harddisc_default_dav = dav[ sc:7, ba:3, du:0, dv:-1 ];
    local_printer_default_dav = dav[ sc:7, ba:1, du:-1, dv:-1 ];
    SRM_default_dav = dav[ sc:21, ba:0, du:8, dv:-1];
```

The device address vector, or DAV, is the data type which describes how a peripheral device is addressed. These constants set up the addressing which is normally used to talk to some standard peripheral devices. Some of the information will be overridden if the peripheral is found at a different address.

- **sc** is the interface select code. Select code 7 corresponds to the built-in HP-IB port at the rear of 9836 family computers. The 9885 disc is connected using a 16-bit parallel interface on select code 12, and a DMA card. The SRM interface is normally set to select code 21.
- **ba** is the HP-IB bus address of the peripheral. Usually an 8290x is addressed as device 0; so is a 9895. The 9134 family of hard discs are expected on bus address 3, and printers on bus address 1.

For the SRM only, **ba** indicates the node number of the SRM interface in a cluster (as opposed to the node number of the Workstation itself).

- **du** selects the drive unit in a multi-drive controller. For instance, a 9895 may have drives 0 and 1. For the SRM only, **du** indicates the unit number within the SRM.

- **dv** selects a particular volume in a multi-volume member of the 7908 (CS/80) disc family.
- **H** is the letter designating the default hard disc. This letter stands for a 9134 micro-Winchester disc drive.

The following statement, found in module `OPTIONS`, governs the byte transfer timeout used by the local printer driver. The timeout, expressed in milliseconds, specifies the maximum time allowed for each byte handshake to complete. A value of zero is a special case, specifying an infinite timeout.

```
const local_printer_timeout = 12000; {milliseconds}
```

The policy of enforcing a timeout on each individual byte works quite well with most HP-IB printers, since they tend not to hold off bus handshakes much longer than the time it takes them to print a single character. However, with printers on other interfaces (notably serial interfaces) we have a different matter. Some serial printers will "buffer up" bytes at high speed until their internal buffer is full, but then will not allow any more transfers until their internal buffer is almost empty. Thus, depending upon the printer's internal buffer size, the maximum time between two bytes being transferred may be the time it takes to print hundreds or even thousands of characters! For these printers, you might consider a timeout of several minutes, or even an infinite timeout.

In general, most HP-IB printers accept hundreds of bytes per second, so you might think that the default 12 second timeout is excessive. We were forced to use this large a number since some low-cost HP-IB printers take 8-10 seconds to execute a full-page formfeed. If you are using a faster printer, you might consider reducing the timeout to 2-3 seconds, so that a real timeout condition will be detected more quickly.

```
const sysunit_list_length = 9;
type sysunit_list_type = array [1..sysunit_list_length] of unitnum;
const sysunit_list =
  sysunit_list_type [ 11,9,7,45,4,44,3,43,3 ];
```

This list governs the selection of a system unit during boot-up. The anticipated significance of the unit numbers is:

Unit #	Default
-----	-----
11	The first volume of a hard disc.
9	A 9885 (drive zero).
7	A 9895 (drive zero).
45	The Shared Resource Manager.
4	Flexible disc drive one of mainframe or external flexible disc peripheral (primary DAM).
44	Drive one flexible disc, secondary DAM.
3	Flexible disc drive zero, primary DAM.
43	Flexible disc drive zero, secondary DAM.
3	Whatever is in unit #3, no directory recognized.

The last choice is provided only for the obscure case where all the other alternatives fail; it leaves the system volume as #3: even though no logical volume was successfully accessed there. Note that the right-hand internal minifloppy drive is drive zero, and the left-hand drive is

drive one. This is opposite the normal practice, eg for a 9895, 8290x or 9121 drive zero is the left-hand drive!

```
const sc_list_length = 3;
type sc_list_type = array [1..sc_list_length] of byte;
const sc_list = sc_list_type [7,8,14];
```

This list governs the select codes which are searched for HP-IB interfaces with recognizable, responding devices. 7 is the select code for the built-in HP-IB port. 8 is a select code usually used for a second HP-IB interface. 14 is the select code usually used for the 98625A high-speed disc interface card.

The select codes are searched in the order they appear in the list (7 first). On each select code, addresses 0,1, ... 7 are polled for devices. In the case of multiple devices contending for an assignment class, say multiple local hard discs, generally the last one polled will be the one assigned.

```
const minimum_volume_size = 1000000;
      maximum_number_vols = 30;
```

These constants govern the partitioning of a hard disc into non-overlapping volumes. No volume will be smaller than one million bytes, and no device is ever partitioned into more than 30 volumes. NB: maximum_number_vols MUST be no greater than 30 !!

```
type multi_volume_option_type = (single_volume,multi_volume,
                                auto_volume);
const multi_volume_option = auto_volume;
```

This option controls the partitioning of hard discs into more than one volume. The Pascal system is the only one which will partition discs into multiple volumes; BASIC is unable to do so.

When multi_volume_option = auto_volume, CTABLE looks at any hard disc attached to the system and tries to determine the structure of the disc. If ALL the directories of the multi-volume partitioning are found, CTABLE will set up the unit table correspondingly. If the first volume is found but one or more other volume directories are absent, CTABLE treats the disc as having a single volume, and marks empty those unit table entries which would otherwise have been used to partition the disc. The result is that if the disc was initialized by BASIC, Pascal and BASIC will both treat it as having a single large volume; but if the disc was initialized by Pascal, Pascal will see multiple volumes and BASIC will see only the first one. BASIC will only be able to access the amount of space allocated to that first volume, and the rest of the disc will be inaccessible to BASIC.

When multi_volume_option = single_volume, Pascal will not partition the disc; it will treat it as a single volume. You should use this selection when the hard disc is accessed by both BASIC and Pascal. This option makes the entire disc available to both languages, albeit with only a single directory.

When multi_volume_option = multi_volume, Pascal will set up the unit table to partition the disc regardless of the existing directory structure on the disc.

Module CTR

This module should not be modified.

Built into it is a lot of knowledge about the supported HP mass storage products, and provides a general structure into which can be inserted information about new peripherals they are introduced.

Each peripheral is assigned a letter designator; these are listed in CTR's export section. In addition there is descriptive information about the size of each type of device, expressed in bytes per track and tracks per medium. The routines in CTR avoid partitioning other than on track boundaries to avoid very inefficient disc access patterns.

Most of the procedures exported from CTR are given a name prefixed with "TEA__". These are the Table Entry Assignment routines. There are TEA__ routines for all the supported mass storage products. Some TEA__ routines are appropriate for an entire family of related mass-storage products.

Utility routines

CREATE__TEMP__UNITABLE allocates in the heap a temporary structure like the real system Unitable. CTABLE makes its assignments to this temporary structure, then uses ASSIGN__TEMP__UNITABLE to copy the final result into the actual system table. NB: ASSIGN__TEMP__UNITABLE will not overwrite any RAM volumes which have been created in the system Unitable. This feature is provided so that if you execute a CTABLE while the system is running, you won't lose files in memory.

SYSUNIT__OK checks to see if a particular unit is blocked, online and has a directory; if so, it is a legal candidate for the system unit.

If you have read the description of the fields of a Unitable entry, you will be aware that two of them are procedure variables which must be initialized to the names of the DAM (Directory Access Method) and TM (Transfer Method or driver) appropriate to the volume and physical device. DAMs and TMs are not part of CTABLE and so would ordinarily be supplied by the linking loader when CTABLE is loaded.

However, there is no guarantee that the DAMs and TMs for a device are present, since they may have been removed from INITLIB or never even installed. Consequently, CTABLE has been programmed to examine the symbol tables kept in memory by the linking loader. If a driver's name is found, it can be used; otherwise, the program avoids references to absent drivers. The routine which searches for link symbols at run-time is called VALUE and is exported from module CTR.

Module BRSTUFF

This module should not be modified.

It exports two routines. `INTERNAL__MINI__PRESENT` determines if there are any 5.25-inch minifloppy drives in the mainframe. `GET__BOOTDEVICE__PARMS` determines what type of device was used for booting and returns the DAV (device address vector) for that device.

Module SCANSTUFF

This module shouldn't be modified.

Its purpose is to inquire from certain disc drives about their size and identification. To do this, the `VALUE` routine (see module `CTR`) is used to find routines which are present only if the driver modules supporting these discs are installed.

Discussion of the Main Body of CTABLE

A lot of details of the behavior of `CTABLE` can be modified by changing declarations such as the select-code list from the `OPTIONS` module. If you want to force some particular assignment, this may be achieved by modifications to the code in the body of `CTABLE`.

After some initializations, `CTABLE` scans the select codes listed in `OPTIONS`. For each select code, and for bus addresses 0 through 7, the program inquires if a device is present. A letter designating the device is returned.

The information about the boot device is extracted. This may be used later in selecting the system unit.

If there are any CS/80 disc drives (7908 family) present, they may have associated "complementary devices", which is a fancy name for the streaming backup tapes. These are also searched out.

Then a temporary `Unittable` is created in the heap. The assignments made as `CTABLE` executes will be made to this temporary table, and only at the end will the real system `Unitable` be updated.

Next, certain standard assignments are made. It is wise not to change these assignments, since programs tend to depend on them. Unit #1 is the CRT, and #2 is the keyboard. #3 and #4 will be 5.25-inch flexible disc drives, or 9121 3.5-inch microfloppy drives. If both internal drive(s) and an external flexible disc drive are present, the internal drive(s) will be used for #3 and #4 unless the external disc was the boot device. This policy gives preference to the higher-performance internal disc drives.

If an SRM interface is present, it is assigned to #5. It may also be assigned to #45 later in the program.

If a 9895 is present, it will be assigned to units #7 and #8. Note that the 9895 should be set to bus address 0, while a 9134 should be set to address 3.

(This is not a requirement; merely a habit which avoids troublesome conflicts. In the Pascal 1.0 system, 9895's were set to bus address zero and 9134's were set to bus address 3. While this is still OK, it's not necessary since Pascal 2.0 scans to locate devices.)

If a 9885 is present, it gets units #9 and #10.

CTABLE can only deal with a single local hard disc, which is found during the HP-IB scanning process. The last one found is retained in the variable HARDDISC_LETTER, and now comes into play in the local hard disc assignment section of the program.

This code is surrounded by conditional compilation directives, because you may wish to not compile it and instead force particular assignments.

CTABLE will normally break a hard disc into more than one volume. As things are arranged (see OPTIONS), no volume will be less than one million bytes and no disc will be divided into more than 30 volumes. The units assigned to these volumes are #11, #12, ... through #40, depending on the number required for the disc. Later, however, volume #11 may be altered to span the entire disc medium and the other entries "zapped" into oblivion. This happens if the disc looks like it was initialized by BASIC, which can only deal with a single volume on a disc. See the discussion of multi_volume_option above.

If a CS/80 backup tape is present, it is assigned to #41.

Next, the alternate-DAM entries are assigned. This allows most discs to be used regardless of the resident directory type.

Units #43 and #44 are alternates for units #3 and #4. Unit #45 is not really an alternate; it is another SRM volume, and may be assigned as the system volume later. If this happens, the OS will have two units on the SRM: one for system files, work files, stream files etc, and another for the "default" working directory. This avoids any possible need to prefix the SRM away from the system volume.

Units #47 and #48 are alternates for units #7, #8 (9895 drive). #49, #50 are alternates for #9, #10 (9885 drive).

We are now about 1250 lines into the CTABLE program, and we come to the templates for "manually" specifying mass storage table entry assignments. These templates are surrounded by conditional compilation directives which cause them to be skipped. To make them effective, one must reverse the sense of the compilation test from \$if false\$ to \$if true\$.

There are templates for the following devices: 8290x; 9895; 9134 A, B and C; 9121; and CS/80 disc types 7908, 7911, and 7912. Each template gives the opportunity to specify DAM, select code, bus address, drive unit, offset in bytes from front of volume to directory, drive letter type, and size of volume. For multiple-volume drives, the templates include a for-loop which calculates how to break up the disc space in the preferred fashion.

Next, the temporary Unitable is copied into the system's Unitable (except that RAM volume entries are not overwritten).

Then the SRM unit entries are prefixed to the appropriate directories. Each workstation in an SRM cluster has an identification number called its "node number", and it is **strongly** recommended that the cluster be configured so that every workstation's node number is unique. CTABLE tries to prefix #45 to a directory called /WORKSTATIONS/SYSTEMnn, where nn is the node number. If no such directory exists, it tries to use directory /WORKSTATIONS/SYSTEM (with no node number). If that doesn't exist either, #45 is "zapped", ie assigned a dummy entry.

This is a rather key mechanism. It allows the workstations in a cluster to have unique configurations. For the normal functioning of the Pascal system, a system volume is required to hold the system library and various system files. If all workstations shared the same system volume, file name collisions would be a real nuisance. CTABLE supports this partitioning, and so

does the overall booting process, allowing for instance a different INITLIB, AUTOSTART and TABLE for each workstation.

The system unit must now be selected. This is done according to the priorities set in the list SYSUNIT__LIST exported from OPTIONS.

Finally, if the system unit is #45 (SRM), unit #5 is also an SRM volume. In that case, #5 is prefixed back to the root SRM directory #5:/ so the root is the initial default volume for the system right after it boots up.

An Example Configuration

This section provides a simple example of how to create your own custom configuration.

One such system where the default auto-configuration may not provide the configuration you want is where you have a Model 26 or 36 (with internal disc drives) and you also want to be able to access an HP 9121 3.5-inch disc. Auto-configuration normally does not assign table entries for both internal drives and external 3.5 or 5.25-inch drives. There aren't enough table entries for both. However, for this system, we will assume there isn't an HP 9885 8-inch flexible disc drive present. This allows entries 9 & 10 to be assigned instead for the external 9121 drives. The actual changes to the configuration program are described below under "Editing CTABLE".

In the example, we assume you have a minimum system: namely, one internal drive, minimum RAM, and the six system discs supplied. In general, this configuration requires the most moving of files, swapping of discs, and reloading of subsystems. If you have a larger mass storage device, you can probably eliminate much of the file copying and swapping of discs. If you work exclusively with a slower mass storage device, but have extra RAM, you may elect to P-load such things as the FILER, EDITOR, and COMPILER.

Mass Storage Setup

To compile CTABLE, you must have the source file (CTABLE.TEXT) and a code file containing the text it imports (INTERFACE) online. Also, you must have mass storage space available for the Compiler to generate various temporary files and the resulting code file (CTABLE.CODE).

CTABLE imports text from several system modules. This text is contained on the LIB: disc in the file INTERFACE. There are several ways to make the import text accessible to the compiler.

- Use the LIBRARIAN to copy the modules in INTERFACE into LIBRARY. The compiler will automatically search LIBRARY if it is online. If you often import system modules, you might want to use this method.
- Insert the compiler directive \$SEARCH in the source text of CTABLE, specifying where to find the file containing the import text (INTERFACE). If you prefer not to have INTERFACE in LIBRARY, you might want to use this method.
- Temporarily redefine the system library file to be INTERFACE instead of LIBRARY.

For the example, the third method was chosen since it makes the fewest permanent changes. The SYSVOL: disc is used as the working volume.

1. Insert the ACCESS: disc and load the Filer by typing: F.
2. Set the default prefix to our working volume. Press P and type:

```
SYSVOL:
```

3. Copy the file INTERFACE from the LIB: disc to our working disc. Insert the LIB: disc, press F (for Filecopy) and type:

```
LIB: INTERFACE, $ [enter]
```

4. Swap discs as directed by the Filer.

5. Copy the source for CTABLE from the CONFIG: disc to our working disc giving the destination file a new name since we are going to modify it. Insert the CONFIG: disc, press F and type:

```
CONFIG:CTABLE.TEXT,NEWCTABLE.TEXT [enter]
```

6. Swap discs as directed by the Filer.

7. Exit the FILER by typing Q.

8. Redefine the system library by using the command interpreter's What command. Press W (for What), B (for liBrary) and type:

```
INTERFACE. [enter]
```

Notice the period.

9. Exit the What command by typing Q.

Editing CTABLE

Insert the ACCESS: disc and run the Editor by typing E. After the Editor is running, insert the SYSVOL: disc and answer the Editor's "File ?" prompt by typing:

```
NEWCTABLE [enter]
```

Recall that you are going to access an HP 9121 by assigning entries for them at units #9 and #10. There are several ways to accomplish this, but the method used here is via the supplied "templates". They are the most straight-forward and the easiest to understand.

The templates are found toward the end of the file, after the comment:

```
{templates for "manually" specifying mass storage table entry assignemnts}
```

You may quickly locate this section by using the Editor's Find command. Press F (for Find) and type:

```
/templates/
```

The cursor will then be at the start of the templates. Normally, you would scroll down to locate the desired template. For this case, however, the desired template is the first one. It looks like this:

```
$if false$ { HP8290X }
  tea_HP8290X( 3, primary_dam, {sc} 7, {ba} 0, {du} 0);
  tea_HP8290X( 4, primary_dam, {sc} 7, {ba} 0, {du} 1);
$end$
```

The `$if false$` and `end` Compiler directives cause the Compiler to unconditionally skip over the text between. Thus the "tea" procedure calls have no effect. You will want to change the "`$if false$`" to "`$if true$`". The "tea" procedures themselves, are defined in the module "ctr". They actually perform the Table Entry Assignments. The reason that you use the "tea" procedures for the HP8290X drives, is that the HP 9121 drives behave just like the HP 8290X drives. You might note that you would also use the HP8290X "tea" procedures for the 5.25-inch drive in the HP 9135 and the 3.5-inch drive in the HP 9133.

The first parameter in the "tea" procedures specifies the unit number you wish to assign. It must be in the range from 1 thru 50. The second parameter specifies the Directory Access Method, or DAM. The DAM specifier is of enumerated type "ds_type". Exported from module "ctr", ds_type is shown here.

```
type
  ds_type = {Directory access method Specifier for local mass storage}
    ( primary_dam,      {either LIF or UCSD, as specified in options}
      secondary_dam,   {the one not selected as primary}
      LIF_dam,         {LIF, regardless of primary/secondary choice}
      UCSD_dam );     {UCSD, regardless of primary/secondary choice}
```

A "tea" procedure has parameters only for those items which are applicable to the device. Furthermore, all parameters are range-checked by the "tea" procedure. While the range-checking cannot guarantee the correctness of your parameters, it can nearly guarantee that your parameters won't ruin the system.

The remaining parameters for all the local mass storage "tea" procedures are device-specific. Most devices will need addressing information such as select code (sc), HP-IB bus address (ba), and disc unit number (du). Parameter descriptions for partitioning hard discs are presented at the end of this example.

Assign entries for two drives on select code 7 (the internal HP-IB) and bus address 2. Logical units 9 & 10 will be used for access via the primary directory access method; logical units 49 & 50 will be used for access via the secondary access method. Our modified template looks like this:

```
$if true$ { HP8290X }
  tea_HP8290X( 9, primary_dam, {sc} 7, {ba} 2, {du} 0);
  tea_HP8290X(10, primary_dam, {sc} 7, {ba} 2, {du} 1);
  tea_HP8290X(49, secondary_dam, {sc} 7, {ba} 2, {du} 0);
  tea_HP8290X(50, secondary_dam, {sc} 7, {ba} 2, {du} 1);
$end$
```

You may leave the templates where they are, or you may move them. However, all "tea" procedure calls must take place between these two statements:

```
{ Create a temporary table & fill it with dummy entries }  
create_temp_unitable;
```

and:

```
{ assign the new unitable and unitclear all units }  
assign_temp_unitable;
```

You may assign and re-assign logical units as many times as desired between the two statements above. When the same logical unit is assigned multiple times, the last assignment performed will be the one that remains in effect.

In the example, if you do not move the templates, logical units #9, #10, #49, and #50 are initially assigned for the HP 9885. However, because the modified templates follow the original assignments, the templates override them.

Quit editing, Save the file, and Exit the Editor by typing Q, S and E.

Compiling and Running CTABLE

1. Load the Compiler by inserting the CMPASM: disc and typing C. Insert the SYSVOL: disc and answer the Compiler's "Compile what text?" prompt with:

```
NEWCTABLE [enter]
```

2. Answer the "Printer listing?" prompt with:

Y for a listing. N for no listing. E for an "errors only" listing (if you have a printer). L for a listing file.

3. Press [enter] to say that the default output file name of "SYSVOL:NEWCTABLE.CODE" is fine.

If you followed the example, you shouldn't have any compilation errors.

4. Press R or RUN to execute the new CTABLE.

Verifying the New Configuration

Generally, the Filer provides the quickest way to verify your configuration. The Volumes command provides a quick sweep of all units. The List command provides a way to test individual units.

Remember that the Volumes command shows only those units which are on-line and which have valid directories. It won't show units with media containing either no directory or the wrong type of directory.

If the first attempt to List a unit fails, the Filer displays:

```
Please mount unit #9  
'C' continues, <sh_exc> aborts
```

Type C. The Filer will then give the reason for failure. A key result is "no directory on volume", which means that the device and medium are accessible, but no directory was found. Other results such as "device absent or unaccessible", "medium absent", or "device not ready" mean that the attempt to read from the device failed.

If you find get "device absent or unaccessible", there may be several possible reasons. A good trick at this point is to eXecute ACCESS:MEDIAINIT on the unit number of interest. For those device types MEDIAINIT recognizes, it will print out the expected device type, plus the addressing information. This is an excellent way to verify the expected configuration, even if the device itself is unaccessible. Don't worry about specifying a device that you really don't want to initialize; MEDIAINIT always prompts for your confirmation before it begins initializing.

Making the New Configuration Permanent

Once you are satisfied with your new configuration and wish to make it permanent, copy the code file to your BOOT: disc. First, however, you should link the code file to itself, in order to conserve disc space.

To link the code file to itself:

1. Invoke the Librarian by inserting the ACCESS: disc and pressing L.
2. Insert the SYSVOL: disc, press I (for Input) and type:

```
NEWCTABLE [enter]
```

3. Press O (for Output) and type:

```
NEWCTABLE [enter]
```

4. Press L (for Link).
5. Press D (to remove the file's Def table).
6. Press A (to link All the modules).
7. Press L (to finish Linking).
8. Press K (to Keep the file).
9. Press Q (for Quit).

Now you are ready to perform the final operations.

To Install the New TABLE

1. Insert the ACCESS: disc and type F (for Filer).
2. Remove the original TABLE file. Insert the BOOT: disc, press R (for Remove) and type:

```
BOOT:TABLE [enter]
```

3. Krunch the BOOT: disc, since your new TABLE file may be larger than the old one. Press K (for Krunch) and type:

```
BOOT: [enter]
```

4. Respond to Crunch directory BOOT: ? (Y/N) with Y.

5. Now copy the new code file from SYSVOL: to BOOT:, giving it the required name. Insert the SYSVOL: disc, press F (for Filecopy) and type:

```
NEWCTABLE.CODE,BOOT:TABLE [enter]
```

6. Swap discs as directed by the Filer.

7. Save your new source file on the CONFIG: disc too. Insert the SYSVOL: disc, press F and type:

```
NEWCTABLE.TEXT,CONFIG:$ [enter]
```

8. Swap discs as directed by the FILER.

9. Clean up the SYSVOL: disc by removing all the files you put there. Use wildcards to save typing. Insert the SYSVOL: disc, press R, ? and [enter].

10. Respond N to the prompt to remove LIBRARY, and respond Y to the prompts to remove INTERFACE, NEWCTABLE.TEXT, and NEWCTABLE.CODE. Respond Y to the confirmation prompt.

11. Exit the FILER by typing Q.

Hard Disc Partitioning

For devices which are partitioned into multiple logical volumes, there are "tea" procedure parameters for logical volume offsets and volume sizes in bytes. In that case, the template will contain a FOR loop, to assign consecutive logical unit entries to the same device, each entry corresponding to a different portion of the medium. The standard way to partition a device with T tracks into N volumes is to allocate T DIV N tracks to each entry, and then add the remaining T MOD N tracks, if any, to the last entry.

While it is suggested that you use the standard partitioning method, there is nothing that forces you to do so. If you like, you may remove the FOR statement, duplicate the "tea" procedure call N times, and specify arbitrary volume offsets and sizes of your choosing for each logical volume. The "tea" procedure checks to ensure that your logical volumes each lie inside the media boundaries. Unfortunately, the "tea" procedure can't check to see if any of them overlap!

In those templates capable of partitioning media, you will find the following line:

```
{ mp := block_boundaries(mp); {override track boundary partitioning}
```

This allows you to use the standard partitioning method, except that the partitioning will occur on 512-byte block boundaries; not necessarily on track boundaries. The "{" character at the beginning of the line makes the line a comment. You invoke the line by deleting the "{" character. Depending upon the media parameters and the number of logical volumes, this may or may not make a difference in how your media actually gets partitioned. This feature is provided

solely for compatibility with discs used with Pascal 1.0. If you don't need it for this reason, don't use it!

All parameters in the templates have typical values for your convenience.

If you get a "value range error" when you execute CTABLE, it probably means that one or more of your parameters is out of range. Don't worry about your system configuration; the old configuration will still be in effect. You can immediately go back to the Editor to try to determine the problem with your new CTABLE.

To find where the value range error occurred, usually the quickest way is to examine the "tea" procedure calls you just modified, and then examine the "tea" procedure itself to see what range it checks the parameters for. However, unless you are a certified wizard, don't modify the "tea" procedure itself!

If you still can't find the source of the error, you can re-compile CTABLE with \$debug on\$. Get a listing from the Compiler too. Then execute CTABLE again. When it terminates with the error again, use the queue (Q) command in the debugger to determine the line numbers of the statements leading up to the error. Also, when you examine the queue, you may need to trace back several line numbers to actually locate the offending statement.

Summary

CTABLE is provided to automatically configure the Pascal workstation to a wide variety of peripheral devices. The automatic configuration can be overridden by modifying CTABLE and substituting the result for the TABLE program on the boot disc or in the boot volume of the SRM.

Chapter 4

SRM Set-Up

Overview

The Shared Resource Manager (SRM) may be the only mass storage for a machine with no local disc drives. This chapter explains how to set up Pascal 2.0 workstations in an SRM cluster. If you are not using an SRM system, you may wish to skip this chapter.

The person who will be System Manager for the SRM cluster is the one who should perform the process described in the next few pages.

In order for your system to work happily with the SRM, every Workstation in the SRM cluster **must** have a unique node number. See the information supplied with the SRM to learn about node numbers.

Booting

If your computer, when turned on with no ROM system installed in the backplane and no discs in the drives, does not identify itself on the CRT as "Bootrom 3.0" or a later version, then your computer must be booted from the internal 5.25-inch flexible disc drive; the SRM can only be used after booting is complete. If your computer IS equipped with a version 3.0 (but NOT 3.0L, which is used with low-cost versions of the 9816 computer) or later Boot ROM, it is possible to boot directly from the SRM.

Boot files on the SRM are found in a directory called SYSTEMS under the the root directory; they have names like SYSTEM_P. This too is explained in the SRM literature.

The System Volume

To allow each Workstation in a cluster to boot up a unique system and have its own system volume, a private directory is established for each node number.

Strictly speaking, this is not always necessary; if a workstation has local high-performance mass storage, it may be desirable to use that as the system volume, and in fact the automatic configuration process will preferentially select high-performance mass storage as the system volume if it is present. However, it doesn't hurt anything to set up unique directories for each workstation, and the following discussion explains how to do so as if everyone would wish to. If you first set things up as explained below, you then have the option to copy frequently used files such as the Editor and Compiler from the SRM onto a volume of local high-performance mass storage. Then when you boot the system those files will be found locally and accessed with correspondingly greater speed.

In the SRM's root directory there should be another directory called /WORKSTATIONS. Under this there should be a directory called SYSTEM, and for each node number "nn" there should also be a directory called SYSTEMnn. For instance, if there are three Workstations on nodes 13, 14 and 15, then in the root there should be:

```
/WORKSTATIONS/SYSTEM
/WORKSTATIONS/SYSTEM13
/WORKSTATIONS/SYSTEM14
/WORKSTATIONS/SYSTEM15
```

Under `/WORKSTATIONS/SYSTEM` should be copies of all the system files, such as the Compiler, Filer, and Editor. These files will normally **not** be used.

Under the private directory for each node should be accessible all the files normally used by the Workstation. For files which don't change, such as the Compiler, it is sufficient to simply have a duplicate link to the original file in `/WORKSTATIONS/SYSTEM` -- there is no need to actually copy such invariant files. The Filer's Duplicate command can be used for this purpose.

Also in a node's private system directory can be the files which "personalize" a Workstation: the system `LIBRARY`, `INITLIB` which controls what drivers and `DAMs` are available, an `AUTOSTART` file, and so forth.

Once this setup is created, booting is a smooth and automatic process. The particular system to be booted is selected by name at power-up. Thereafter, the Workstation looks for necessary files in the directory with its node number. If a required file can't be found, default is taken to `/WORKSTATIONS/SYSTEM`, and if something crucial is still missing, the boot may fail. (It will complain on the console.)

If you boot from the SRM, your system volume will be unit `#45` (prefixed to your private directory `/WORKSTATIONS/SYSTEMnn`) and your default volume will be `#5` (another SRM volume, prefixed to the SRM root directory).

If you booted from local mass storage rather than the SRM, your system files will probably have been found on that local mass storage. The SRM is still available through units `#5` and `#45`.

One more thing: in order to run properly, there must be one more special directory called `TEMP_FILES` under `WORKSTATIONS`. All temporary files are created in this directory, and are removed when no longer needed. If you don't create this directory, the first workstation to need it will do so. Should the create fail, an error is reported. Consequently the directory `WORKSTATIONS` should **not** be write-protected unless directory `TEMP_FILES` has already been created.

Most users will also want a private directory for their default volume. Typically one creates a directory called `USERS` under the root, and within `USERS` a private directory for each individual. After booting, use the Filer to set the current working directory for your unit `#5` to your private directory. This keeps the root directory from getting cluttered. Your own private `AUTOSTART` stream file is very nice for this purpose.

SRM Initialization

This section tells what to do the first time you set up the first Pascal workstation with the SRM software. It should NOT be repeated for every workstation you set up! Once this procedure is complete, the SRM will be accessible any time you boot up your workstation.

It is assumed that your SRM hardware has been installed and tested as prescribed in the SRM literature. There are four parameters which are set when the SRM configuration is initially created. Appropriate values for these parameters when using Pascal workstations with the SRM are:

IOBUFFERS: At least five per workstation in the cluster;
 (e.g. 40 buffers for 8 workstations).

DISC BUFFERS: Fifty is a good choice.

TASKS: Two is enough.

FILES: Allow for ten or twelve open files per workstation
 in the cluster; one hundred is a nice round number.

For the following procedures you will need:

- The wiring chart and node number assignments which were prepared when designing your SRM installation.
- The six original Pascal system discs:

BOOT: Must be installed at powerup time.

SYSVOL: Contains (part of) the system library.

ACCESS: Contains Filer, Editor, disc initialization
 utility etc.

CMPASM: Contains the Pascal Compiler and Assembler.

CONFIG: Contains optional resident software which
 may be installed in the boot files.

LIB: Contains optional software such as Graphics
 and device IO which may be installed in the
 system library.

These discs may be supplied in several different formats, depending on what mainframe computer and mass storage options you ordered. Most likely the discs you have are 5.25" flexible diskettes.

- A local mass storage device which can read these six discs.
- Two blank magnetic discs like the six originals.

Installing the SRM software is not a hard or complicated operation, but it is important that the steps outlined be followed in exact detail. Since you are less likely to make mistakes if you understand what's going on, here is an outline of what you will do.

1. Determine how much memory your computer has. In the process which follows, it is helpful to have two disc drives instead of just one; or to have a fixed disc such as the HP 9134. If your computer doesn't have additional mass storage, then it **MUST** have at least 512K bytes of read/write memory, which will be used as mass storage.
2. Boot up your computer using the original supplied BOOT disc. If you will be using some read/write memory as mass storage, it is particularly important that you use the original disc or an exact copy; if things have been added to the BOOT disc, there may not be enough memory available.
3. Use the MEDIAINIT program to initialize the blank discs. This must be done before a directory can be written on the disc. Then use the Filer program to make an empty directory on one disc, and a copy of BOOT on the other.
4. If you have "enough" memory and don't have two disc drives or a hard disc, reserve some memory to use as a pseudo-disc. This is called "making a RAM volume".
5. This sounds tricky but isn't. On the BOOT disc is a file called INITLIB (Initial Library) which contains, among other things, the software which enables your computer to "talk" to peripheral devices. Such software is generically called "drivers". We will use the LIBRARIAN to create a new INITLIB file containing the stuff in the original INITLIB and the drivers required to talk to the SRM. The SRM drivers are on the CONFIG disc.
6. Use the Filer to replace INITLIB on the BOOT disc with the new one.
7. Reboot the computer using the new BOOT disc. At this point the computer is able to talk to the SRM.
8. Create the required directories on the SRM, then transfer the various system files such as the Compiler out to the SRM.
9. Reboot. You're in business. Other machines on the cluster can use copies of the BOOT disc you've just prepared. If you wish to include still more drivers in the INITLIB of some systems, see the section "How to add driver modules to INITLIB".

OK, here we go.

Determine available memory

Here is what to do if you don't know how much memory is in your computer. If the machine is on, turn it off, along with any disc drives to which it is connected. Open the doors of the built-in flexible disc drives if there are any. If already connected to the SRM, remove the cable connected to the 98629A Resource Management interface in the back. Now turn the computer on. After going through self-test, the CRT will display the amount of available memory.

If you have at least 524000 bytes, there is "enough" memory to proceed with only one disc drive. Otherwise you must have a second disc drive to proceed. If you have a second disc drive, use it even if you have "enough" memory.

You should also note what Boot ROM revision is installed in your computer. If when powered up as just described, the Boot ROM identifies itself on your CRT as "Boot ROM 3.0" or a later revision (but NOT 3.0L, which is used with low-cost versions of the 9816 computer), your computer is capable of booting directly from the SRM.

Reconnect the SRM cable. Turn on any peripherals connected to your computer.

Boot-up Using the Original BOOT Disc

Turn off the computer, then follow the normal sequence described in the Pascal Language System User's Manual to boot up. This basically involves inserting the BOOT disc into the right-hand flexible disc drive (the only one on a 9826) and turning on the power. However, if you have never booted up the Pascal system before, you should read the first few chapters of the User's Manual now!

Note that the "right-hand flexible disc drive" is an imprecise term; what is really meant is "drive number zero". Drive zero is the right-hand drive in a 9836, or the only drive in a 9826. But if your system uses an 8290x series disc or a 912x series disc, drive zero is the **left-hand drive**.

After booting, you will be at the "Command" level; at the top of the screen you should see a line beginning with the word "Command:" and the cursor will be blinking at the right end of this line.

Prepare the Blank Discs

A disc must be initialized before data can be stored on it; this process, also sometimes called "formatting", stores on the disc information identifying the areas where data may later be written. Initialization is discussed in the Pascal Language System User's Manual, under "Backing Up Your System Discs".

You need to initialize both the blank discs you will be using.

Remove the BOOT disc (or SYSVOL if, during booting, the computer asked you to insert it) and put the disc labeled ACCESS in the right-hand disc drive. Then type:

```
[execute] MEDIAINIT [enter]
```

Here [execute] means press the key labeled "EXECUTE", or "EXEC" for a 9816. [enter] is of course the "ENTER" key. Be sure to type the word MEDIAINIT in capitals! The program will be loaded and start running, and ask:

```
Volume id?
```

You answer:

```
#3 [enter]
```

If the disc has been used previously and already has a directory on it, MEDIAINIT will tell you the name of the existing directory. Then you will see:

```
WARNING: the initialization will also destroy:
#43 <no dir>
Are you sure you want to proceed?
```

Don't worry about this. Answer yes by typing:

```
Y
```

The next prompt is:

```
Interleave factor [1..15] (default is 1)?
```

The particular default interleave factor will vary with the type of disc drive you are using. In any case accept the default by pressing:

```
[enter]
```

The program will now initialize the disc and inform you when it is done. To restart the program you need only put in another blank disc and press U (for User restart).

Finally you should use the Filer to put directories on the discs just initialized. Actually, MEDIAINIT puts a directory on each, but it is always called "V3" or whatever is the number of the disc drive used during initialization. So we will put some more useful names.

Put the ACCESS disc in the same drive you loaded MEDIAINIT from, and press the F key while at the Command prompt level. When the Filer has loaded, it will show its own prompt at the top of the CRT, beginning with the word "Filer:".

Use the Zero command by pressing the Z key. The Filer responds

```
Zero directory (NOT valid for SRM type units)
Zero what volume?
```

You answer:

```
#3 [enter]
```

It asks:

```
Destroy V3: ? (Y/N)
```

Answer:

```
Y
```

It asks:

Number of directory entries (80) ?

Tell the Filer that 80 directory entries is fine by pressing:

[enter]

It asks:

Number of bytes (270336) ?

or some other number which is the size of the disc medium. You answer:

[enter]

It asks:

New directory name?

You answer:

NEWLIB [enter]

Actually you could pick some other name; 6 or less characters, capital letters recommended. The Filer responds:

NEWLIB: correct ? (Y/N)

You answer yes:

Y

The disc spins, then the Filer replies:

Volume NEWLIB zeroed

It is wise to label the disc you have just made. Write the name on a label **before** applying the label to the disc; sharp instruments are likely to damage the disc. Don't touch the exposed surface of the magnetic medium under any circumstances!

It is not necessary to put a directory on the second disc; we will get one out there by copying the BOOT disc verbatim. Note however that the BOOT disc has 8 directory entries instead of 80. This disc will be quite full but have only a few files; there is no need to waste space with unused directory entries.

While still in the Filer program, insert the original BOOT disc in the drive you have been using and type F for the Filecopy command. The Filer responds:

Filecopy what file?

You answer:

#3,#3 [enter]

The Filer will read the contents of the disc (which may take a few moments) and then prompt:

```
Please mount DESTINATION in unit #3
'C' continues, <sh-exec> aborts
```

Replace BOOT with the second blank disc and press the C key. The Filer will tell you when the copy is complete.

Now exit the Filer and return to the Command prompt by typing Q.

If necessary, make a RAM volume

If you determined that you have "enough" memory and must use some memory for mass storage, the following steps are necessary. At the Command prompt level, press:

```
M
```

The computer responds:

```
*** CREATING A MEMORY VOLUME ***
```

```
What unit number?
```

You answer:

```
#50 [enter]
```

It asks:

```
How many 512 byte BLOCKS?
```

You answer:

```
520 [enter]
```

It asks:

```
How many entries in directory?
```

You answer:

```
8 [enter]
```

It finishes:

```
#50:      (RAM:)  zeroed
```

This has reserved 266,240 (520*512) bytes of memory to use as a mass storage device. It is like having a disc drive with a disc named "RAM" inserted in it.

Make a new INITLIB

Now we will make a new Initial Library which includes the drivers required for the SRM.

Insert the ACCESS disc into the drive you have been using and press L to load the Librarian. When you see the Librarian's prompt line at the top of the CRT, use the O (Output) command to specify the name of the file the Librarian will be creating.

If you **are** using a memory volume, type the sequence:

```
O RAM:INITNEW [enter]
```

If you **are not** using a memory volume, you have an auxiliary disc drive. Put disc NEWLIB in that drive and type:

```
O NEWLIB:INITNEW [enter]
```

If you use an auxiliary drive, you **must not remove** the NEWLIB disc until the end of this step, after you have exited from the Librarian.

Next tell the Librarian to take input from the INITLIB file on the BOOT disc. Remove ACCESS from the drive and insert the BOOT disc copy you made earlier. Then use the I command as follows:

```
I BOOT:INITLIB. [enter]
```

Be sure to type the period after the word INITLIB in this command. The Librarian will respond by showing INITLIB as the name of the input file. Near the bottom of the CRT you will see a line which says:

```
M input Module: KERNEL
```

Press the T key to transfer this module to the output file. After a few moments, the name of a new module (KBD) will appear. Each time a new module name appears, press T to move it to the output file. You should continue copying modules until the name LAST appears; **DON'T COPY MODULE LAST** yet. Here is the list of modules you will transfer before you get to LAST:

```
KERNEL
KBD
KEYS
CRT
BAT
CLOCK
DEBUGGER
PRINTER
DISCHPIB
AMIGO
IODECLARATIONS
HPIB
DMA
REALS
ASC_AM
WS1.0_DAM
TEXT_AM
CONVERT_TEXT
LIF_DAM
```

Now you must get the required SRM drivers from the CONFIG disc and include them in the output file. First close the input file by typing:

```
I [enter]
```

Then remove BOOT and insert CONFIG into the drive, and type:

```
I CONFIG:DATA_COMM. [enter]
```

Don't forget the period after the name. When the module name DATA__COMM show up near the bottom of the screen, type:

```
A
```

This tells the Librarian to transfer all the modules in the file. Then use the I command again to pick up the SRM input file, again being sure to type the period after the file name:

```
I CONFIG:SRM. [enter]
```

Again transfer all by typing:

```
A
```

Now you will swap discs again. Type:

```
I [enter]
```

before removing the CONFIG disc, then put in BOOT once more. Again type:

```
I BOOT:INITLIB [enter]
```

When module KERNEL shows up near the bottom of the screen, select module LAST instead by typing:

```
M LAST [enter]
```

then transfer it by typing:

```
T
```

You now have all the modules in your new library. "Keep" it and then quit the Librarian by typing:

```
K Q
```

Now we will use the Filer to save a spare copy of the library just created on NEWLIB (the second disc you initialized in step C). Insert disc ACCESS and press F to call up the Filer.

If you **are** using a memory volume, insert NEWLIB (the second disc which was initialized in step C) in the drive and type:

```
F RAM:INITNEW.CODE,NEWLIB:$ [enter]
```

to copy the file onto the disc. This makes a permanent copy of the results of step E.

If you **are not** using a memory volume, you may still wish to make a backup copy of the new library.

Replace INITLIB on the New BOOT Disc

If you didn't use the Filer at the end of the last step, load it now by inserting ACCESS and pressing F. Then put in the BOOT disc copy made in the previous step. Press R for the Remove command:

```
Remove what file?
```

Answer with:

```
BOOT:INITLIB [enter]
```

Again, use capital letters! Note also that there is no period after the file name this time. Then pack all the remaining files on the disc to make the maximum amount of room for the new INITLIB: press the K key. The Filer answer:

```
Crunch what directory?
```

You answer:

```
BOOT: [enter]
```

Don't fail to type the colon after the volume name! The Filer will then say

```
Crunch directory BOOT ? (Y/N)
```

Answer:

Y

It says:

```
Crunch of directory BOOT in progress
DO NOT DISTURB!!
Crunch completed
```

If you interfere with the disc before the Crunch operation completes, you will ruin the data on the disc. You will certainly have to recopy it from the original BOOT and you may have to re-initialize it.

Now when the Crunch is finished, you can move the INITNEW library file created in step E onto the new BOOT disc. At the same time, we will rename it INITLIB.

What you should do next depends on whether you're using a RAM volume or not. If you sent your INITNEW library to a memory volume in step E, follow the next paragraph; otherwise skip ahead to "If you are not using a memory volume".

If you **are** using a memory volume, then:

Insert the BOOT disc copy you have been working with and press F for the Filecopy command.

```
Filecopy what file ?
```

Answer:

```
RAM: INITNEW.CODE, BOOT: INITLIB [enter]
```

The Filer will tell you when the file has been copied.

If you **are not** using a memory volume:

Insert disc NEWLIB and press F for the Filecopy command.

```
Filecopy what file?
```

Answer:

```
NEWLIB: INITNEW.CODE, BOOT: INITLIB [enter]
```

The Filer will suck up INITNEW.CODE and then ask you to:

```
Please mount DESTINATION volume BOOT
'C' continues, <sh-exc> aborts
```

You should remove the NEWLIB disc and insert the BOOT copy you have been working with. Then press C. The Filer will write out the new file.

Depending on how much memory is available in your machine, the Filer may ask you to swap the NEWLIB and BOOT discs several times. Follow its instruction precisely. It will tell you the name of the disc to insert each time.

When it finishes, you have a BOOT disc which contains the SRM drivers.

Each Pascal workstation in the cluster must boot using an INITLIB which has the SRM driver software installed. You may wish to make copies of the disc you've just created for each workstation. The disc can be copied using the Filer sequence F#3,#3 [enter] just as we did in step C when copying the original BOOT disc.

Reboot using new disc

Presumably your SRM controller has already been installed according to the supplied instructions; the shared disc has been initialized, and the SRM root directory exists. If your SRM has a printer, you should have also configured a spooling directory. At this point you are ready to begin talking to the SRM. Use the new BOOT disc you have just created. Turn off your computer, insert the new BOOT disc, and reboot.

Create the Directories and Files Required on SRM

Insert the ACCESS disc in drive #3 and press F to execute the Filer. When the Filer prompt appears, press V to list the volumes on-line.

If the SRM is not recognized by your system, you should recheck the SRM installation and verify that the interface in the computer you are using is recognized by the SRM. Also verify that the 98629 interface card is set to select code 21 (unless someone has altered the default select code provided by the IO configuration program TABLE). See the "System Manual for Shared Resource Management", HP part number 09826-90080.

Now create the required directories. If the SRM has already been running with some other systems connected, such as 9845 or 9836 BASIC, some of these directories may already exist. To see the directories which already exist, enter the following Filer commands:

```
L#5:/ [enter]
```

In following the steps below, obviously you should skip the steps which create directories which already exist on your SRM.

To create directory WORKSTATIONS, use the following Filer sequence:

```
M
```

The Filer responds:

```
Make file or directory (F/D) ?
```

You want to make a directory:

```
D
```

The Filer responds:

```
Make directory (valid only for SRM type units)
Make what directory?
```

You answer:

```
#5:/WORKSTATIONS [enter]
```

Be sure to type this name in capital letters! If the root directory was protected with one or more passwords, at this point the Filer would report: 'Error: invalid password'. In that case, you need to find out the required passwords from whoever initialized the SRM disc or installed the passwords. To create this directory, you need Write access rights in the root directory, and possibly Manager rights if they were specified. For instance, if the password for Write access is PLEASE, you would specify #5:/.<PLEASE>/WORKSTATIONS in the step above. Alternatively, you might use the main volume password by specifying #5<VOL_PASS>/WORKSTATIONS . The Filer should reply:

```
Directory is 'WORKSTATIONS' correct ? (Y/N)
```

You answer:

```
Y
```

The directory is created, then the Filer announces:

```
Directory WORKSTATIONS made
```

If the computers in the SRM cluster have a Boot ROM which is able to boot from the SRM (Boot ROM 3.0 or later revision not 3.0L, which is used with low-cost versions of the 9816 computer), you will also want to create a directory called SYSTEMS in the root. Repeat the steps just given, but instead specify that you want to create directory #5:/SYSTEMS .

Next, create directory SYSTEM under WORKSTATIONS. This is where the master copy of all system programs such as the Compiler will be stored. To reduce the amount of typing involved, we will make the current working directory for unit #5 be the newly created WORKSTATIONS directory. Type P for the Prefix command. The Filer responds:

```
Prefix to what directory ?
```

Answer:

```
#5:/WORKSTATIONS [enter]
```

The Filer will respond:

```
Prefix is WORKSTATIONS:
```

Now if you don't specify a unit number in Filer operations, the system will assume you are referring to directory WORKSTATIONS. To create SYSTEM, the sequence is

```
M
Make file or directory (F/D) ? D
Make what directory? SYSTEM
Directory is 'SYSTEM' correct? (Y/N) Y
Directory SYSTEM made
```

Also under WORKSTATIONS create directories called SYSTEMnn, where nn is the node number for each workstation in the cluster. You can see why we said each node number should be unique! For example, create SYSTEM05 for the workstation at node 5. Note that two digits are required if the first digit is zero.

Finally, under WORKSTATIONS you should create a directory called TEMP_FILES. This is only necessary if you plan to write-protect WORKSTATIONS.

You are now at the last stage! It is time to move the required files out into the new directories. First prefix the current working directory to #5:/WORKSTATIONS/SYSTEM using the Filer sequence:

```
P#5:/WORKSTATIONS/SYSTEM [enter]
```

Then insert the BOOT disc in the drive you have been using and copy all the files on it into the current working directory. Use F (the Filer's Filecopy command).

```
F
Filecopy what file?
```

You answer:

```
BOOT: =, $ [enter]
```

The Filer will copy the files one after another. Then repeat the operation for each of the following discs:

```
SYSVOL:
CMPASM:
ACCESS:
LIB:
CONFIG:
```

After this is done, the SYSTEM directory contains the entire Pascal 2.0 Workstation software.

Now you need to make these files available in the private SYSTEMnn directory of each workstation. For each such system directory, use the Filer's Duplicate Link command (D):

```
Duplicate link (valid only for SRM type units)
Duplicate or Move ? (D/M)
```

You want to duplicate links rather than move the files ; this will allow each directory to "see" the files. Press the D key. The Filer will ask:

```
Dup_link what file?
```

Answer:

```
?, #5:/WORKSTATIONS/SYSTEMnn/$ [enter]
```

The questionmark causes the Filer to ask if you want each file transferred. Answer "Y" for every file except AUTOSTART and SYSTEM_P. Of course you should substitute a two-digit node number for nn each time. The Dup_link operation is very fast. It reports each file as the links are made.

The last detail is optional. If any of the workstations in the SRM cluster have Boot ROM revisions 3.0 or later (but NOT 3.0L, which is used with low-cost versions of the 9816 computer) and will be expected to boot from the SRM instead of using local mass storage, you need to put a

copy of the boot image file in directory SYSTEMS under the root (ie in /SYSTEMS, not in /WORKSTATIONS/SYSTEM). The boot image file is on the BOOT disc, but a copy was already made in the paragraphs above. The Dup__link command can move the file to a different directory. Type D, then

```
Duplicate link (valid only for SRM type units)
Duplicate or Move ? (D/M) M
Dup_link what file?
#5:/WORKSTATIONS/SYSTEM/SYSTEM_P,#5:/SYSTEMS/$ [enter]
```

That concludes the required SRM software setup. Now any workstation using the BOOT disc you have created will be able to access the SRM via logical units #5 and #45. If a workstation has high performance local mass storage such as a fixed disc, that workstation's system volume will be on the local mass storage; otherwise the SRM directory #45:/WORKSTATIONS/SYSTEMnn will be the system volume.

It is advisable to also create a private working SRM directory for each user, in addition to the SYSTEMnn directories for each workstation. Typically a user will then use unit #45 for his system volume and #5 will be prefixed to his working directory. A good way to set this up is to create:

```
/USERS
```

in the root directory, and then

```
/USERS/CHARLIE
/USERS/SUE
```

and so forth for each user.

Chapter 5

Programming with Files

File Naming Conventions

The definition of HP Pascal tries to minimize the pain of moving Pascal programs from one operating system to another by requiring the use of string values to specify the names of files and certain other information such as passwords and access rights.

In Pascal 2.0, the allowable syntax of a file name depends on the type of directory in which the file resides. The underlying file support is structured to allow programs to work properly regardless of the directory organization(s) being used, but the syntax of file names is defined by the directory.

File Specifications and File Names

There is a difference between a file SPECIFICATION and a file NAME. A file name is a character string which is the external identifier by which a file is designated in a disc directory. A file specification is a character string which consists of the file name and several other optional items: volume name, path name, passwords and size specifier. Not all these items are allowed by every Directory Access Method or under all circumstances; for instance, path name and passwords are only used with the Shared Resource Manager's hierarchical directory organization.

Syntax of a File Specification

The syntax of a legal file specification is given by:

```
filespec ::= [volumeid] [pathname] filename [sizespec]
          ::= volumeid
```

In this notation, items between braces "[" and "]" are optional; quoted items appear literally. The definition just given means that a filespec may appear in one of two forms. The first form consists of an optional volumeid followed by a colon, then an optional pathid, then a filename which is not optional, then an optional sizespec. The second form consists just of a volumeid.

Syntax of a Volume Identifier

The volumeid selects one of up to 50 logical units known to the file system. If no volumeid is present, the volume used is the "default volume" selected by the Filer's Prefix command. Otherwise the volume is specified in one of two ways:

```
volumeid ::= "#" integer [ password ] ":"  
          ::= name [ password ] ":"
```

In the first case, the integer is a two-digit number from one to fifty; for example, #23: is a volume id. In the second case, the name is a sequence of characters. The length of the name and allowable characters depend on the particular directory organization used by the logical unit. For mass storage devices, the volume name is actually stored on the disc itself so it can be identified whenever it is inserted into a drive. For devices which have no directory, such as printers, the volume name is an arbitrary one supplied by the TABLE configuration program at boot-up time.

Example volumeids of the second form are MYSYS: and PRINTER: Volumeids may be 6 characters long in LIF directories, 7 characters long in Workstation 1.0 (UCSD-compatible) directories, and 16 characters long in SRM directories. Use only upper-case letters and digits for volume names.

A LIF volumeid of all blanks (6 ASCII spaces) will not be recognized.

In the case of a logical unit connected to a Shared Resource Manager, the volumeid takes a special meaning. The notation #5: refers to the current working directory of volume number five; the notation #5:/ refers to the root directory of the SRM to which volume number five is connected. The current working directory for any SRM volume is selected by the Filer's Prefix command.

On the other hand, if the logical unit does not have a hierarchical directory, then the two volumeid notations have the same meaning. This is the case for all local mass storage devices.

Syntax of a Path Name

Directory path names are only allowed when specifying files on SRM logical units. The syntax of pathnames is:

```
pathname ::= [ "/" ] { directoryname [password] "/" }

password ::= "<" word ">"

directoryname ::= filename
               ::= "."
               ::= ".."
```

The use of curly braces "{" and "}" indicates that the stuff between them may occur zero or more times. As you can see, there are two special directory names allowed with the SRM. The name "." (a single period) refers to the current directory somewhere along a path to a file of an SRM logical unit. The name ".." refers to the parent of the current directory. Other file names occurring in a path name are directories along the path to the one which contains the file being specified.

Passwords are sequences of up to 16 characters, which govern the access rights to a file or directory. They are given to a file either at creation time, or by use of the Filer's Access command.

Note that a pathname doesn't appear by itself; it appears as part of a file specification, with the file name after the path name.

Examples of path names are:

/	Denotes the root directory.
/.<PASS1>/	Denotes root, using password "PASS1".
/USERS/ROGER/	Denotes directory ROGER in USERS, which is in root directory.
HERE/THERE/	Denotes directory THERE, found in HERE
../THERE<PASS2>/	Directory THERE, found in the parent of the current working directory.

A path name together with a volumeid might appear as:

```
#5:/WORKSTATIONS/SYSTEM13/
```

Occasionally there is need for a volume password, which is a case not covered by the above syntax. You may use either of the following forms:

```
#5<volpassword>:/dirname1/dirname2/filename
#5:<volpassword>/dirname1/dirname2/filename
```

That is, the volume password may either immediately precede or follow the colon separator.

Syntax of File Names

A file name is just a sequence of characters. The Directory Access Methods allow all printable characters. However, the following characters have significance either in Filer commands or in the overall specification of files under various Directory Access Methods (such as path name in hierarchical directories), and therefore should be avoided in file names:

- sharp '#'
- asterisk '*'
- comma ','
- colon ':'
- equals '='
- question mark '?'
- left bracket '['
- right bracket ']'
- dollar sign '\$'
- less than '<'
- greater than '>'

Control characters (ordinal less than 32) and blanks are removed altogether by the File System before the name is ever presented to any Directory Access Method.

File Types Derived From File Names

The type of a file is determined when it is created, and is derived from a suffix (the last characters of the file name). Once the file type is determined, a type code is recorded in the directory, and changing the file name won't change its type.

Suffix	File type
-----	-----
.ASC	LIF ASCII text file
.TEXT	WS 1.0 / UCSD compatible text file
.CODE	Pascal 2.0 object code
.BAD	File covering bad area of disc
.SYSTEM	Boot image file
<none>	"Data" file (assumed FILE OF <TYPE>)

File Names (LIF Directory)

The LIF Directory Access Method (DAM) generally allows any ASCII character to be used in a file name. This is contrary to the HP LIF Standard, which states that file names must be composed only of upper-case letters, digits, and the underscore '_' character. Note that uppercase and lowercase letters are distinct. File names stored in LIF directories are always exactly 10 characters; they may be blank-padded by the DAM if necessary.

The LIF DAM recognizes only UPPERCASE suffixes.

The 10-character file name length would be a very severe restriction when four or five characters are required for a suffix. To ease this problem, the LIF DAM performs a transformation on the file name which compresses the suffix if one is present. The transformation occurs automatically when a LIF directory entry is made, and it is reversed automatically before the file name is ever presented to any program or to the user.

This "black magic" is usually completely transparent to the Pascal user, although its effects may be seen when a LIF directory is examined from the BASIC language system. It sounds complicated and dangerous, but in practice it is very smooth. Most people would never notice it if they weren't told.

Here is how LIF DAM changes a name before putting it into the directory:

- Look for a standard suffix. If there is none, the file is a data file and the name is used unchanged unless it is too long. If it is longer than 10 characters an error is generated.
- If a suffix is found, it is removed from the name but the dot '.' delimiter is left. If the resulting name is longer than 10 characters, an error is generated.
- If the trimmed name is not too long, the dot is replaced by the first letter of the suffix (e.g. 'A' for '.ASC').
- If the name is now less than 10 characters long, it is extended by appending underscores '_' to 10 characters.

Using this algorithm, we would have the following examples:

'A.ASC'	==>	'AA_____'
'charlie'	==>	'charlie '
'123456789.TEXT'	==>	'123456789T'
'GollyGeeet'	==>	rejected because it would be confused with transformation of 'GollyGeee.TEXT'

The reverse transformation is fairly obvious:

- If the 10th character is a blank, do nothing; otherwise,
- Remove all trailing underscores.
- Compare the last non-underscore to the first letter of each valid suffix. If a match is found, remove that letter from the file name and append a dot '.' followed by the full suffix.
- If no suffix match is found, use the original file name.

File Names (WORKSTATION 1.0 Directory)

The Workstation 1.0 DAM allows file names of up to 15 characters including the suffix. Any lower-case letters are transformed to upper-case, so that 'a.text' and 'A.TEXT' denote the same file.

File Names (SHARED RESOURCE MANAGER)

The SRM itself allows almost any file name. Names shorter than 16 characters are padded with trailing blanks. Two names are reserved: all blanks, and all nulls (ASCII zero). The Pascal 2.0 system removes blanks and control characters from the file name.

The Pascal 2.0 SRM Directory Method takes the "<" character to denote the beginning of a password. All characters up to the next ">" character are part of the password, so that <<<<<<<<> is a (poorly chosen) password. Passwords may be up to 16 characters long.

File Size Specification

The last, optional part of a file specification is the file size specifier. If present, its syntax is:

```
sizespec ::= "[" integer "]"
          ::= ".*"
```

This specification only takes effect if a new file is being created; otherwise it is ignored (e.g. RESET).

In the first form, the integer gives the number of 512-byte blocks to be allocated to the file. For instance, [100] would cause allocation of 51,200 bytes.

The second form: [*] specifies that the file is to be allocated either (half of the largest free space) or (the second largest free space), whichever is larger.

If no size specifier is present when space for a new file is being allocated, the largest free area is assigned to the file.

For files stored in the SRM, the first extent allocated to the file will be contiguous and of the size specified if possible.

Operations

This section describes how to program using the Pascal file operations. It discusses the creation and disposition of files and the basic operations on file data.

Pascal Primitive File Operations

- Opening and closing files:

REWRITE	Open file for writing.
RESET	Open file for reading.
OPEN	Open file for random (direct) access.
APPEND	Open file to extend (append new data).
CLOSE	Close file and optionally discard it.

- Sequential file operations:

READ	Read next component.
WRITE	Write next component.
GET	Invalidate current file component.
PUT	Place current file component.
<file>^	Validate and access file buffer.
EOF	Predicate - test end-of-file.

- Direct access operations:

SEEK	Position to specified file component.
READDIR	Read specified file component.
WRITEDIR	Write specified file component.
POSITION	Returns current component number.
MAXPOS	Returns highest allowable component number.

- Operations on textfiles:

READ	Input data editing.
READLN	Pass end-of-line.
WRITE	Output data editing.
Writeln	Output end-of-line.
PROMPT	Output data editing.
OVERPRINT	Write over current line.
EOLN	Predicate - test end-of-line.
PAGE	Write form-feed.

Grouping the operations functionally, produces:

- The following operations put the file into WRITE Mode:

```
REWRITE
OPEN
APPEND
SEEK
PUT
WRITE
WRITEDIR
WRITELN      {see the section on TEXT files}
F^           {if the file is already in WRITE Mode}
```

- The following operations put the file into READ Mode:

```
RESET
GET
READ
READDIR
READLN {see the section on TEXT files}
```

- The following operations put the file in LOOKAHEAD Mode:

```
F^      {unless the file was in WRITE Mode}
EOF      {unless the file is open for random access}
EOLN     {see the section on TEXT files}
READs    {of multi-character objects from TEXT files, such
          as strings, PACs, integers, reals, enumerated types,
          and booleans.}
```

File Position

In order to understand the three modes a file can be in, we need to take some time to discuss the **file pointer** and the **file buffer**, F^{\wedge} .

Associated with each open file is file pointer. This pointer can be thought of as a marker indicating how much the file has been read or written. The file pointer starts at the beginning of the file when the file is opened with **RESET**, **REWRITE**, or **OPEN**. The file pointer is set to the end of the file if the file is opened with **AKPPEND**. The file element pointed at by the file pointer is called the **current component**. Each time you read from a file, the current component is fetched. Each time you write to a file, the new information becomes the current component.

The components of a file are numbered sequentially from 1 to N , where N is the number of components in the file. The **file position** is a number from 1 to $N+1$ which usually corresponds to the position of the file pointer.

The Buffer Variable

Each file has associated with it a special variable called the **buffer variable** or the **file window**. This is a variable of the same type as the components of the file. It is referred to as F^{\wedge} where F is the file identifier. For example, if F is a **FILE OF INTEGER**, then F^{\wedge} is an integer variable. The buffer variable is usually associated with the current component of the file.

File States

Every file which is open is in one of three states or modes at any given time depending on what was the most recent operation on that file. The file state has to do with whether you are reading or writing the file and whether you have referenced the **buffer variable**, F^{\wedge} . The three states are **WRITE Mode**, **READ Mode**, and **LOOKAHEAD Mode**.

If the file is in **WRITE Mode**, F^{\wedge} has no special meaning other than as a variable, and referencing it causes no I/O to take place. This is the mode in which you normally assign to F^{\wedge} , i.e.,

```
 $F^{\wedge} := \dots ;$ 
```

in preparation for a **PUT** statement. If you assign from F^{\wedge} , i.e.,

```
 $\dots := F^{\wedge} ;$ 
```

in this mode you will get unpredictable results.

The **READ Mode** is also called the **LAZY I/O state**, because in this mode the buffer variable refers to the current component of the file, but the file system does not fill it until the first time it is referenced. In this mode you normally assign from F^{\wedge} in order to read the next component of the file.

If the file is in **READ Mode**, referencing F^{\wedge} causes the current component to be fetched from the file and placed in the buffer variable. When this is done, the buffer variable is full and the file goes into the **LOOKAHEAD Mode**. Once the file is in the **LOOKAHEAD Mode**, F^{\wedge} may be referenced as many more times as desired but no more I/O will be done.

The **LOOKAHEAD Mode** is so called because we have **peeked** at the current component without having advanced completely past it. In actuality, the current component has been read into F^{\wedge}

and the file pointer has advanced to the following component. However, the file system pretends that the current component hasn't been fetched yet. In this state the POSITION function returns a value corresponding to the component in the file buffer, which is 1 less than that corresponding to the true file pointer. Also, in this state, READ(F, V) will assign the value of F^ to V instead of reading the next component of the file. On the other hand, if a write were done in this state, it would write the component at the true file pointer, and the POSITION function would appear to advance by 2 instead of 1!

Creating New Files

A file is initially created by the REWRITE, OPEN, or APPEND operations. However, OPEN and APPEND are usually applied to existing files. These standard procedures each may take one, two or three parameters:

```
REWRITE (filevar)
REWRITE (filevar.name)
REWRITE (filevar.name.thirdparam)
```

Here "filevar" is the name of a Pascal file variable; "name" is a string which is the system identification of the file; and "thirdparam" is an optional string which is used with Shared Resource Manager files to control shared access to the file (see subheadings "Third Parameter to Reset, Rewrite, Open", "Concurrent File Access", and "Access Rights" below). The name string may include information about the file size, and an SRM pathname to the relevant directory.

When a new file is first created, it is considered "temporary", and it will remain so until it is CLOSED with a specification that it be LOCKed into the disc directory. Such temporary files don't conflict with other files of the same name. A new file created by REWRITE, OPEN, or APPEND will be thrown away when the program terminates unless the program takes explicit action.

The allowable file name syntax depends on the Directory Access Method (DAM) being used; this subject is discussed under "File Naming Conventions", below. However, all file names may have appended to them a specification of the size of the file, which the DAM may use at file creation time to allocate space. The size specification may take the following forms.

- No size specification. The file will be allocated the largest available block of space for contiguous-file DAMs (LIF and Workstation 1.0 directory organizations), or an indeterminate amount of space for the SRM. Example: CHARLIE.TEXT
- [*] on end of file name. The file will be allocated the greater of (second largest free block, half of largest free block) for contiguous-file DAMs, or an indeterminate amount of space for the SRM. Example: SUSANNAH[*]
- [nnn] on end of file name, where "nnn" is a positive integer. The file will be allocated nnn blocks of 512 bytes each for contiguous-file DAMs, or an indeterminate amount by the SRM. Example: EXACTLY[1000] which gets 512,000 bytes.

It is permissible to create anonymous files by specifying no file name, for example: REWRITE(F). Note however that there is no way to request a specific file size for an anonymous file; REWRITE(F, '[500]') is not acceptable because there is no file name preceding the size specifier.

The REWRITE, OPEN, and APPEND primitives do not necessarily create a new file. Whether they do depends on whether a file already exists with the given name, and whether the file variable is already associated with some physical file by virtue of a previous opening operation.

REWRITE(F) (with optional 2nd and 3rd parameters)

If F was already open at the time of REWRITE and no filename is specified, the same physical file is referenced. If a filename is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit CLOSE is actually a CLOSE(F, 'NORMAL') -- see below -- and so the file will not necessarily be saved. The file is positioned to its beginning, and any data it contained is discarded. Thus one way to overwrite the content of an existing file is to open it for reading via RESET, then REWRITE it.

If the file variable F is not already associated with a physical file (that is, F is not presently open), a new file is created and opened for writing. If a file name and size are specified, they will be applied. The new file created is temporary until it is CLOSED, and in fact is distinct from any existing file of the same name!

APPEND(F) (with optional parameters)

If F was already open at the time of APPEND and no filename is specified, the same physical file is referenced. If a filename is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit CLOSE is actually a CLOSE(F, 'NORMAL') -- see below -- and so the file will not necessarily be saved. If F was open and no filename is given, then APPEND positions to the end of the file and re-opens it for writing. Any data written will get tacked on to the file; the original content remains valid.

If F is not already open and no file name is given, a new temporary file is created and the behavior is like REWRITE.

If F is not open and a file name is given, APPEND searches for an existing file of that name. If one is found, position to the end and prepare for writing; if none is found, create a new temporary file.

Restrictions on APPEND

APPENDING to text files is not allowed in the Pascal 2.0 implementation. It only works for data files (file of <type>).

If the file is in a volume with a WS 1.0 directory organization, it may not be possible to APPEND. For this directory type, APPEND is only allowed if there happens to be free space on the disc immediately following the current end of the file.

If the file is in a volume with a LIF directory, it may not be possible to APPEND. Space for the file is initially allocated as requested (see File Size Specification, below). When the file is closed, its logical end of file and physical end of space are both recorded. Subsequent APPENDING will always be allowed until the physical space is exhausted; after that, APPENDING will only work if there happens to be free space on the disc after the current end of file.

OPEN(F) (with optional parameters)

Opens a file for random (direct) access, allowing both reading and writing. The file is positioned to its beginning.

If F was already open at the time of REWRITE and no filename is specified, the same physical file is referenced. If a filename is specified, the current file is closed and the physical file

specified by the second parameter is referenced. This implicit CLOSE is actually a CLOSE(F, 'NORMAL') -- see below -- and so the file will not necessarily be saved.

If F is not open and no file name is given, a new temporary one is created. If a file name is given matching an existing file, that file is used; otherwise a new file is created.

Disposing of Files

A program terminates the association between a file variable and a physical file with the CLOSE standard procedure. The call may specify that the file is to be purged (deleted from the directory), locked (made permanent), or purged only if it is a temporary file.

CLOSE(F, 'SAVE')	Both do the same thing; the file is made permanent in the volume directory. Does nothing if file is anonymous.
CLOSE(F, 'LOCK')	
CLOSE(F)	Both do the same thing. If the file is already permanent, it remains in the directory. If it is temporary, it is removed.
CLOSE(F, 'NORMAL')	
CLOSE(F, 'PURGE')	The file is removed from the directory whether or not it was permanent.
CLOSE(F, 'CRUNCH')	The end-of-file marker is set at the current file position; data beyond this position is lost.

Opening Existing Files

To open an existing file, you must give a file name parameter to the OPEN, APPEND or RESET standard procedures.

RESET(F,'filename')

Opens an existing file for reading, and positions F to the beginning. If F was already open and no file name is specified, the file to be read is the one which was open. Otherwise, the file system searches for an existing file of the specified name and reports an error if none is found.

RESET(F) with no file name specifier will fail unless F is already open.

OPEN(F,'filename')

APPEND(F,'filename')

OPEN and APPEND search for the named file, and if one is found, the association will be with that physical file. But note that if no file is found, a new temporary one will be created; see comments about file creation, above.

Note that OPEN(F) and APPEND(F) without a file name will create new files unless F was already open.

REWRITE(F,'filename')

When **REWRITE** specifies the name of a file which already exists, a new temporary file is created. All output data goes to this new file instead of the old one. At the time the file is closed, the old one is purged and the temporary file is renamed. This prevents destruction of the old file in case the program terminates prematurely.

To get rid of the old file first, open it with **RESET** and then do a **CLOSE(F, 'PURGE')**.

Sequential File Operations

In Pascal there are two classes of file: **TEXT** and **DATA**. Files of type **TEXT** are so declared in the Pascal program:

```
var F: text;
```

Text files are best thought of as lines of characters, separated by end-of-line designators of some sort. They are intended to represent human-legible text material such as documents.

Data files are files of some component type. They are ordered sequences of variables, all of the same type. The type may be a predeclared type like **INTEGER**, or some user-declared type:

```
type
  rec = record
      name: string[50];
      socialsecurity: integer;
  end;
var
  ss: file of rec;
```

A file of **char** is not the same thing as a text file, because no lines are distinguished in the file of **char**.

This section is about data files; the discussion of text files is below. In the discussion, **F** denotes a file variable; **T** is the type of its components; and **V, V1, V2 ..** are variables of type **T**.

READ(F,V)

If **F** is open for reading (by **RESET** or **OPEN**), then this standard procedure will store into variable **V** the current component of **F** and advance to the next component. Note that **READ(F, V1, V2, V3)** is equivalent to three **READs** in a row.

WRITE(F,V)

If **F** is open for writing (by **REWRITE**, **APPEND** or **OPEN**) then the value of **V** is written as the current component of **F** and **F** is advanced to the next component. **WRITE(F, V1, V2, V3)** is allowed.

F[^]

The file variable name can be referenced as a pointer. It points to the "current" component of the file; that is, if **F** is a file of **T**, then **F[^]** is a variable of type **T**. **F[^]** is called the "buffer variable" of **F**. (This logical buffer is distinct from the physical device buffer!)

Note that for files being read, **F[^]** is always valid unless end-of-file has been reached. This means that whenever a reference is made to **F[^]**, the file system will ensure that the current component has been fetched from mass storage and is available in the buffer variable.

HP Pascal specifies the use of "lazy evaluation", which simply means that the buffer variable is not filled until the program references it.

PUT(F)

PUT and **WRITE** are related operations. To output data using **PUT**, first store into the buffer variable the value to be written, then call **PUT**:

```
F^ := V; PUT(F);
```

This sequence is equivalent to **WRITE(F, V)**.

Note that it isn't enough to just store into **F[^]**; you must also **PUT** the value. For instance:

```
F^ := V1; F^ := V2; PUT(F)
```

will store into the file the single value **V2**. Also, if you fail to **PUT** the last component before closing the file, the last component will be lost.

GET(F)

This is the complementary operation to **PUT**, used for input. It throws away the current component value and advances the file to the next component. The sequence:

```
V := F^; GET(F);
```

is like **READ(F, V)**. Note that since **F[^]** is always valid unless end-of-file has been reached, the first component of **F** is available immediately after **F** is opened; there is no initial **GET**:

```
RESET(F, 'CHARLIE'); {open file for reading}
V := F^;             {take 1st component}
GET(F);              {advance to 2nd component}
```

EOF(F)

This predicate returns true if the file is positioned and end-of-file, else false. At end-of-file, the content of **F[^]** is undefined and any attempt to **READ(F,...)** will cause an error.

EOF is true if **F** is not currently open, or if it is open for write only.

Direct Access (Random Access) Files

Files of DATA (not TEXT) may be accessed directly, that is, a program can specify that it wants to read or write the *n*-th record in the file without scanning through the records in sequence. A file must be opened with the OPEN standard procedure to allow direct access.

The components of a direct access file are numbered sequentially, with the first being number one. (Note that there is no acknowledged standard in this area; for instance, UCSD Pascal numbers the first component of a direct access file as record zero. All HP Pascal implementations work as described herein.)

When a file is OPENed, it is positioned at the first component. If sequential IO operations are performed, the file components will be accessed in ascending order. There are several ways to randomly access the *n*-th record.

READDIR(F,N,V)

The read-direct standard procedure positions F to component N of the file, and then reads the value into variable V. Subsequent READ calls would receive records N+1, N+2 and so on. READDIR(F,N,V1,V2,V3) is equivalent to the sequence:

```
READDIR(F,N,V1);
READ(F,V2);
READ(F,V3);
```

WRITEDIR(F,N,V)

The write-direct standard procedure positions F to component N of the file, and then writes value V. Subsequent WRITES will place values in components N+1, N+2 and so on. WRITEDIR(F,N,V1,V2,V3) is equivalent to:

```
WRITEDIR(F,N,V1);
WRITE(F,V2);
WRITE(F,V3);
```

SEEK(F,N)

As with the other direct-access procedures, file F must be OPENed (for both read and write). SEEK positions F so that the next call to GET or PUT will fetch or place component N.

```
OPEN(F,'CHARLIE');
SEEK(F,100);
GET(F);
V100 := F^;
```

This definition is certainly counter-intuitive in that the program **must not** do an initial GET after opening the file, but **must** after SEEKing.

SEEK works most smoothly (in the most natural fashion) if used with READ and WRITE:

```
SEEK(F,N);  
READ(F,V);
```

POSITION(F)

This function returns an integer value which is the number of the next component which will be read or written. If the buffer variable F[^] is full, POSITION returns the number of that component.

MAXPOS(F)

This function returns an integer value which is the number of the last component which has ever been written into the file. Note that the component must have been written; merely SEEKing out to some far component is not enough to cause the maximum position limit to be extended.

Textfile Input and Output

A TEXTFILE is composed of variable-length lines of characters. It differs from FILE OF CHAR in that the lines are separated by end-of-line marks. Pascal 2.0 supports three different text file representations. Text files are the basis of human-legible input and output (as it says in the Pascal standard). This means that they are used for "formatted" IO such as printouts.

Declaring a TEXT File

A text file must normally be declared in the following way:

```
VAR F: TEXT;
```

All text files must be declared except the two standard files INPUT (corresponding to keyboard) and OUTPUT (which sends its output to the CRT). These two files, if used, must be listed in the main program header:

```
PROGRAM X (INPUT,OUTPUT);
```

However, they must NOT also be declared in the body of the program.

In addition, there are two other "standard" system files which may be used, called KEYBOARD and LISTING. If these two files are used, they must appear both in the program heading and in a VAR declaration, as follows:

```
PROGRAM X (INPUT,OUTPUT,KEYBOARD,LISTING);  
VAR  
  KEYBOARD,LISTING: TEXT;  
BEGIN  
  . . .  
END.
```

Don't worry about why INPUT and OUTPUT must not be declared yet KEYBOARD and LISTING must be; that's how it is. Note also that the four standard files are automatically opened by the Operating System before the program runs -- they do not generally appear in RESET or REWRITE statements, although they may be closed and re-opened if necessary.

KEYBOARD and INPUT both take characters from the keyboard; the difference is that characters read from INPUT are echoed to the CRT, while those read from KEYBOARD are not. The file LISTING is opened to PRINTER:LISTING.ASC which is the standard system printer. (Note that since PRINTER: is normally an unblocked volume, the file name part of the specifier is ignored. On the other hand, if PRINTER: is a mass storage volume, the file name is significant. It's a good habit to include a file name even when going to unblocked volumes.)

Representations of a TEXT File

The way lines of characters will be represented in a text file is determined when the file is originally created.

If the file name given in the REWRITE which creates the file ends in the suffix .ASC, the file representation used is LIF (Logical Interchange Format) ASCII. In this representation, each line is preceded by a signed 16-bit length field telling how many characters are in the line. In this representation, there is no restriction on what characters may appear in the line.

If the creation file name ends in the suffix .TEXT, the representation used is known as "Workstation 1.0 format". This format is compatible with the UCSD Pascal P-system textfile representation ("UCSD Pascal" is a trademark of the Regents of the University of California), and may be used as a non-HP interchange format. For instance, text files produced by the HP-87 P-system implementation can be read in this format. (In fact, Pascal 2.0 can both read and write HP-87 P-system discs if the directory offset field in the appropriate Unit table entry is correct).

The WS1.0 format precedes lines with a leading-blank compression indication, and terminates each line with an ASCII carriage-return character. Leading blank compression occurs when a line is written, and the compressed blanks are expanded when the line is read. When using this format, don't write the characters NUL (zero), CR (13) or DLE (16). Moreover, note that TABs (9) are not expanded! Generally it is wise to avoid writing any characters with ordinal value less than 32 into WS 1.0 textfiles.

If the textfile is created anonymously (no file name given) or with any other suffix, the "data file" representation is chosen. In this case, a carriage return denotes end-of-line, and all other characters are passed through uninterpreted.

Note

The representation of a text file is **not** a function of the directory format being used. An ASCII file may be present in a WS1.0 directory, or a WS1.0 text file in a LIF directory.

The LIF ASCII representation can only be used if the LIF ASCII Access Method module is installed in your system's INITLIB boot file. The WS 1.0 format can only be used if the UCSD Access Method module is installed in INITLIB.

If the required Access Method is not installed, the system will choose the "data file" representation regardless of file name suffix.

Formatted Input and Output

The use of `WRITE`, `WRITELN`, `READ`, and `READLN` to write formatted output to text files is described in many Pascal reference documents and will not be repeated here, except to take note of the behavior when reading and writing character strings.

HP Pascal supports two forms of character string, generically referred to as PAC (for Packed Array of Char) and `STRING`. A PAC is a variable whose type specification is of the form:

```
TYPE T = PACKED ARRAY [1..N] OF CHAR;
```

Where `N` is some integer constant. The lower bound of a PAC must be one in HP Pascal, although Pascal 2.0 allows any arbitrary lower bound if the `$UCSD$` Compiler compatibility mode is enabled.

When a string literal value is assigned to a PAC, if the string is shorter than the declared length then the string is blank-padded to the declared length. Thus if a 5-character literal is assigned to a 10-character PAC, the last 5 characters of the PAC will get blanks. This same behavior occurs on input of a PAC value (see below).

When a PAC is written to a text file, all `N` characters are put out unless a shorter field specification is given in the `WRITE` statement:

```
TYPE
  PAC = PACKED ARRAY [1..10] OF CHAR;
VAR
  S: PAC;

S := 'abcde';      {PAD WITH 5 TRAILING BLANKS}
WRITE(F,S);       {WRITE 10 CHARACTERS}
WRITE(F,S:5);     {WRITE FIRST 5 CHARS}
WRITE(F,S:15);    {WRITE 5 BLANKS, THEN ALL 10 CHARS OF PAC}
```

A `STRING` is a variable whose type specification is of the general form:

```
TYPE S = STRING [N];
```

Where `N` is a constant between 1 and 255 giving the maximum allowable length of the string. `STRING`s differ from PACs in having an implicit variable length. Usually the length of a string is the length of the last string value assigned to it, although string length can be explicitly manipulated by the standard procedure `SETSTRLEN`.

When a `STRING` variable is read from a text file, its length is set to the length of the incoming string (see below). When written, a `STRING` takes the number of characters specified by its current length.

Reading a STRING or PAC from a Textfile

When a string is read from a textfile, its length is usually determined by an end-of-line marker. If the incoming line length is less than or equal to the string length, then if the receiving variable is a PAC it is padded with trailing blanks as necessary to fill it completely; if it is a STRING, its length is set to indicate how many characters were read.

If the entire string is filled before end-of-line is reached, the read operation ceases. No error is reported, and the next character read will be the one following the last one read.

When reading strings, an end-of-line must be explicitly passed by READLN. If you repeatedly read into a string while positioned at an end-of-line marker, you will keep getting back an empty STRING or a PAC of all blanks. The approved way to read long lines into short strings is:

```
WHILE NOT EOF(F) DO
  BEGIN
    REPEAT
      READ(F,S);
      ... {process the piece of string}
    UNTIL EOLN(F);
    READLN(F);
    ...
  END;
```

You should be aware of one other fact about end-of-line handling in READs: reading strings or PACs is the only situation in which end-of-line is not automatically "swallowed". The Standard states that when EOLN(F) is true, the value of F[^] is a blank. When reading a number, for instance, end-of-line is not treated differently from any other blank in the character stream of the input text file.

The Third Parameter

The optional third parameter to the standard file opening procedures is used at the time of file creation to control concurrent access to files and to specify file access rights via passwords. This parameter is a character string whose syntax conforms to the following definition:

```
thirdparam ::= [ concurrencyword ]
            ::= [ passwordlist ]
            ::= concurrencyword ";" passwordlist

concurrencyword ::= "SHARED"
                ::= "EXCLUSIVE"
                ::= "LOCKABLE"

passwordlist ::= capability [ "," capability ]

capability ::= password ":" accessrightlist

accessrightlist ::= accessright { "," accessright }

accessright ::= "READ"
              ::= "WRITE"
              ::= "PURGELINK"
              ::= "CREATELINK"
              ::= "SEARCH"
              ::= "MANAGER"
              ::= "ALL"
```

Note that in the passwords themselves, upper and lower case letters are distinct. Examples of third parameter strings:

```
'SHARED'
'EXCLUSIVE;MYSECRET;MANAGER'
'LOCKABLE;R:READ,W:WRITE'
'Charley:ALL'
```

SRM Concurrent File Access

Three modes of access to shared files are allowed:

- **EXCLUSIVE:** No concurrency. Only one workstation may open the file at a time. This is the default for all files opened on the SRM.
- **SHARED:** No controls. The file may be opened by any number of workstations for both reading and writing. This is particularly dangerous for multiple writers since, for performance reasons, some local buffering is done in each workstation. Different buffers may overlap parts of the same file, and may not contain identical data! Shared file users will not be aware of changes in actual end-of-file induced by the actions of other users.

Shared files are primarily intended to be used by multiple readers.

- **LOCKABLE:** This mode provides strict concurrency interlocking by means of file operations **LOCK**, **WAITFORLOCK**, and **UNLOCK**. The file must be locked to perform any operation on it; only one reader/writer may access the file at a time. A series of operations or a single operation be performed while it is locked. The initial lock obtains the necessary physical file status information from the SRM, and unlocking updates all the status information on the SRM as well as flushing buffers. Thus when the file is unlocked, its contents are always complete and consistent.

Note

Shared access is allowed concurrently with lockable access and may circumvent the integrity provided by the locking mechanism.

The user-callable routines which support locking are provided in the library module **LOCKMODULE**, which is in the standard system **LIBRARY**. To use them, the program must **IMPORT LOCKMODULE**. The specifications for these routines are:

- **FUNCTION LOCK (ANYVAR F:FILE): BOOLEAN;** This function returns true if the lock succeeded, or false if the lock failed because the file was already locked. Other IO errors such as file not open generate an error which may be trapped using **TRY/RECOVER** (see System Programming Language Extensions).
- **PROCEDURE WAITFORLOCK (ANYVAR F:FILE);** This procedure sends the SRM a request to lock the file, and then waits until it is confirmed.
- **PROCEDURE UNLOCK (ANYVAR F:FILE);** This procedure releases the file so another workstation can lock it.

The file locking capabilities are primarily intended for data files (Pascal file of <type>) which are opened for random access using the standard procedure `OPEN`. Suppose `F` is a file which is not already open. The cases are:

- `OPEN (F, 'filename')` The existing file is opened for exclusive access. The open will fail if the file is already open by some other workstation. This is the default.
- `OPEN (F, 'filename', 'EXCLUSIVE')` The existing file is opened for exclusive access. The open will fail if the file is already open by some other workstation.
- `OPEN (F, 'filename', 'SHARED')` The file is opened for shared access. Any number of workstations may have the file open `SHARED` at the same time. They may read or write -- there is no synchronization.
- `OPEN (F, 'filename', 'LOCKABLE')` The file is opened in such a way that no access is permitted unless the file is first put in the locked state. Any number of workstations may have a file open `LOCKABLE` at a time, but only one workstation may have the file locked.

`REWRITE`, to a file which is already open within the program performing the `REWRITE`, simply repositions the file to its beginning and sets it up for writing.

If `REWRITE` specifies the name of a file which does not exist, a new file of that name is created and used.

If a physical filename is given and a file of that name exists, the existing file is opened with whatever concurrency specification (shared, exclusive) was given in the `REWRITE`. If no physical file exists, one of the given name is created and opened with the requested concurrency specification. This action is in addition to the creation of the temporary file, and helps prevent interference by other workstations.

Surprising effects may occur if two workstations `REWRITE` the same physical file concurrently. The one closed last will remain in the SRM directory.

Note that `REWRITE(F, 'LOCKABLE')` is probably not a sensible operation. However, it does not generate an error.

SRM Access Rights

Passwords can be used to restrict the types of access allowed to a file (on the SRM a directory is also a file). They can be set by the Filer's Access command, or at the time a file is created. Passwords can control the following six types of access:

- READ
- WRITE
- SEARCH
- CREATELINK
- PURGELINK
- MANAGER
- ALL

Any access rights for which no password is specified belong to the set of public capabilities which are granted to any workstation opening the file without specifying passwords.

The word ALL denotes the six access types collectively. When an ALL password exists, there are no public capabilities. The ALL password allows any file operation to be performed.

SEARCH capability is required on all directories along the pathname to a given file.

RESET requires READ access to the file.

Both READ and WRITE capability are required if the file is opened by calls to OPEN or APPEND.

To REWRITE an existing file, any passwords in the file specification (second parameter to REWRITE) are used only to purge the old file. However, one of the three capabilities READ, WRITE, MANAGER must also be granted to open the old file before purging it. The new file created by REWRITE will have the passwords specified in the third parameter; until this new file is closed, any operations may be performed on it.

WRITE capability on the directory in which it resides is required to CLOSE 'PURGE' a file, in addition to the SEARCH capability needed to open the file and PURGELINK capability on the file.

To CLOSE 'LOCK' a file, WRITE capability is required for the directory, in addition to the SEARCH capability needed to open the file.

If a password with MANAGER capability is used to open a file, any file operations may be performed, since the manager password would allow the access types to be changed. For example,

REWRITE(F, 'FILE1', 'A:ALL') Gives no public capabilities.

REWRITE(F, 'FILE1', 'M:MANAGER') All capabilities except MANAGER are public. This allows any file operations to be performed, but the manager password 'M' is required to change or set passwords.

Debugging Programs Which use Files

The file system uses the TRY-RECOVER and ESCAPE mechanism in its normal internal operations. For instance, when opening a file several escapes may occur internal to the FS or driver calls. These "errors" don't get back out to the user program.

But if the Debugger is used on such a program and error trapping is enabled, the Debugger will stop the computer on each internal escape. This can be very confusing to a user! The clue that this is happening is that the line number displayed by the Debugger in the lower right corner of the screen doesn't change during the FS call.

The most common escape codes generated in this fashion are -10, 2080 and -26. You can suppress the Debugger's activity on these codes with the Debugger command:

```
ETN -26,2080,-10
```

Chapter 6

The Booting Process

Introduction

Pascal 2.0 is a single-task system in which user programs and code modules, as well as most system capabilities appear as extensions to the Operating System kernel. This section describes the components of the system and gives a high-level view of how they fit together. The system booting process is also described.

You cannot make the best use of this material unless you have used and become familiar with the Pascal system. The Compiler reference material in your Pascal Language System User's Manual is practically required reading, with special attention to the discussions of:

- The Pascal MODULE construct
- System Programming Language Extensions
- How Pascal Programs Use the Stack

Concepts of Linking and Loading

All object code files produced by the Compiler, Assembler, or Librarian are called "libraries". A library contains a directory, describing one or more modules of object code. In the context of libraries, the word "module" denotes any of the following:

- The output of one invocation of the Assembler.
- A unit of object code produced by compiling one Pascal MODULE.
- A program (something with a start-address).
- A linked combination of any of the above, produced by the Librarian.

Note that if you compile a program containing two Pascal MODULES, the result will be a library containing three distinct object code modules.

The format of object code modules is described elsewhere; for now you need to bear in mind certain facts. Modules in this system are always relocatable, never absolute. Each module consists of a global data segment, and one or more code segments. Both code and data are relocated (assigned final locations in memory) when the module is loaded. Normally code is emitted so that its relocation base is zero, which simply means that if the code were loaded, unchanged, starting at byte zero of memory, it would run properly.

Code is never loaded at address zero; in fact, there is ROM at address zero in the 9836. It is possible to use the Librarian to change the relocation base of a module to any address desired; this technique can be used to produce the logical equivalent of an absolute code module.

However, this is not done, with a single exception described later. Instead, the linking loader which is always resident in the system performs all relocation as needed.

Not only are modules relocatable, they are also normally unlinked. This means that as a module resides on mass storage, it contains references to addresses EXTERNAL to itself which must be satisfied (filled in with final values) at load time. Even a "linked" module, which has been passed through the linking process of the Librarian, may still contain unsatisfied references which were not filled in by the linking operation.

All such references must be completed before the object code can execute properly; final linking is performed by the linking loader. To satisfy external references, the loader follows a specific search pattern. First it searches other modules in the library being loaded. Then it checks modules which have previously been loaded. Last, it looks in the system LIBRARY file. If, to load module "A", the loader finds that "B" in the system library is also required, the "B" will also be loaded automatically.

How is the linking loader able to link a newly loaded module to others which have been previously loaded and reside in memory? Tables are kept of all the symbols defined in all the modules which are loaded. Every such symbol (corresponding for instance to the name of a module, an exported procedure or a program start address) has a value known to the loader for as long as the module defining the symbol remains resident. The linking loader can even hook up a new module to a module which is currently executing! This is "dynamic linking" is in fact the "induction rule" by which the system constructs itself at boot time.

By the way -- the tables in memory are searched in the order, most-recently-loaded first. This means a module may override other, previously loaded symbols.

Overview of the Booting Process

The booting operation occurs in several phases, which will be described briefly at first, then again in more detail.

When the computer is first turned on, it is under control of the "Boot ROM". This read-only memory resides at address zero, and its first few bytes contain the address of the first executable instruction of the Boot ROM itself together with an initial stack pointer value. The 68000 always powers up by setting its Program Counter (PC) and Stack Pointer (SP) registers to the values found in this ROM. Actually there are several versions of the Boot ROM; as of this writing, version 3.0 is the latest. Versions 3.0 and later identify themselves to the user when the machine is turned on.

The code in the ROM executes a certain amount of self-test programming, then looks around for an operating system. For 3.0 and later versions, the search pattern is a subject in its own right! Later versions of the Boot ROM are large programs which can boot from almost any HP mass storage product. First the Boot ROM tries to find a "soft" (RAM resident) system on various mass storage devices such as the built-in mini floppy drive or a Shared Resource Manager. Failing that, it looks for a hard (ROM resident) system such as BASIC or HPL. If several candidates are found, the operator is given the option to pick one.

The Pascal "soft" system supplied to you is the file `BOOT:SYSTEM_P`, that is, on the volume called `BOOT`: it is the file called `SYSTEM_P`. This is an absolute load-image of the bare minimum core of the OS, containing the linking loader and some support routines. There are no mass storage drivers, the ones in the Boot ROM being used for the nonce. The absolute load-image of the loader was created by us using the `B (Boot)` command of the Librarian.

In this system there is no "kernel" in the closed, absolute sense of an operating system such as UNIX. Rather, the system has an "open" design which allows modules of code providing new capabilities to be added to the system -- while it is running. The linking loader is a sort of "induction rule" which allows the system to grow gracefully. Still, we do use the word "kernel" in this document, meaning roughly the set of modules providing a minimum environment.

The loader begins execution by completing construction of the operating system. This is done by loading the "initialization library" `INITLIB` from the mass storage where `BOOT` was found. During this process you will see the message,

`Loading 'INITLIB'`

on the CRT. As modules are loaded, they are bound into the OS by the dynamic linking process mentioned above. `INITLIB` contains such useful items as the Debugger, IO drivers and the floating-point arithmetic package. You can use the Librarian to examine for yourself the contents of `INITLIB`. The modules are loaded in order of appearance in the library. Several are programs -- they have a start address. After the loading is complete, each program is executed once. Programs in `INITLIB` are referred to as "installation code"; their purpose is to properly initialize variables or steal storage which will be used by the `INITLIB` modules.

By the way, you can delete certain modules to make `INITLIB` smaller, or add modules of your own. You mustn't change the order of the ones supplied, nor link them together (which would result in the loss of the start addresses of installation code. More information on this subject can be found under the heading, "Library Management". Once `INITLIB` is loaded and the installation code executed, the IO driver subsystem has found and identified all the interface cards, although no examination has been made of peripheral devices.

The last piece of installation code in `INITLIB` is a program called `LAST`, which loads and executes two programs called `STARTUP` and `TABLE`. `STARTUP` is loaded before `TABLE` if it resides

on the boot device; otherwise STARTUP is loaded from the system volume after TABLE executes.

TABLE configures the OS so that the fifty "logical units" of the Pascal file environment are correctly related to the "physical units" attached to the IO interface cards. The general subject of peripheral configuration is covered in a previous chapter.

The logical units, designated #1: through #50; are examined in detail in the File System discussion. Essentially they are represented as an array of records called the "Unitable". Each Unitable entry tells such information as the name of the unit and what Directory Access Method (DAM) and Transfer Method (TM) must be used when accessing files on the unit.

To properly initialize the Unitable, the loader now executes the program called TABLE. This configuration program is user alterable; in fact it must be altered and recompiled if you wish to create a non-standard peripheral configuration. It also selects the system volume. You will see the messages,

```
Loading 'STARTUP'  
Loading 'TABLE'
```

displayed on the CRT at this time.

After TABLE executes, a complete environment exists in which any Pascal program is executable. The loader executes the previously loaded STARTUP program. This could be any Pascal program at all, for instance one you write. The one we supply is referred to as the Command Interpreter; it displays the outer-level Command prompt, loads or executes modules at your command, and traps and reports errors. Our Command Interpreter never stops, but if you supply your own STARTUP program and it ever terminates, the system will display the message:

```
SYSTEM FINISHED, RESET TO RESTART
```

How the Boot Files are Chosen

Various products are derived from the Pascal kernel, and these derivative products sometimes need to be able to load "their own" specialized versions of INITLIB, TABLE and STARTUP without interfering with the normal Pascal system. This is particularly true in the Shared Resource Manager environment, where there may be many applications present in a node's directory. Many of these applications may look like stand-alone, bootable systems.

So the kernel needs to be able to chose different libraries. It does so based on the name of the file being booted.

If the file name is SYSTEM_P then the standard Pascal system files INITLIB, TABLE and STARTUP are chosen.

If the file name is SYSTEM_xxx where xxx is some three-character suffix, the chosen file names are INITxxx, STARTxxx and TABLExxx.

If the file name is SYSxxxxxxx the chosen file names are INITxxxxxxx, STARTxxxxxxx and TABLExxxxxxx. The seven-character suffix is only useful when booting from the Shared Resource Manager, since normal boot discs are in LIF format and only allow 10 character file names.

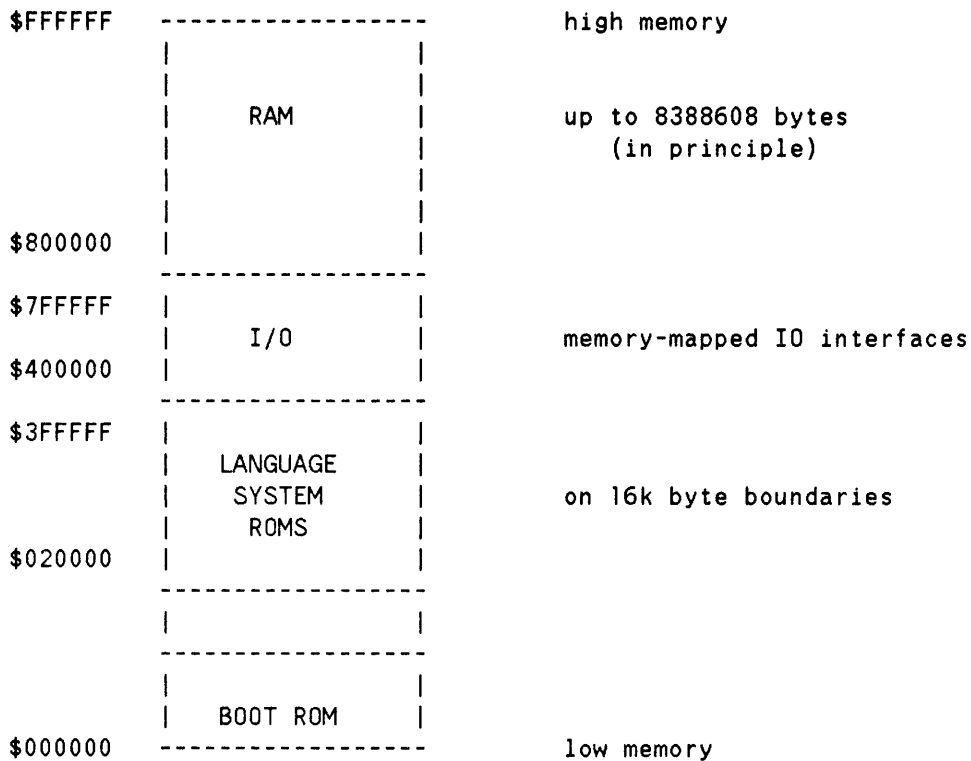
Memory Map Development

To understand booting in more detail, you need to visualize the memory map of the computer as it develops from power-up through the entire booting process.

The 68000 processor has 23 word-address lines called BA1 through BA23. For byte operations, two control lines BUDS (byte upper data strobe) and BLDS (byte lower data strobe) indicate which byte(s) of the word are affected. Thus there is a 24 bit address or 16 megabyte address space. On the other hand, the CPU uses 32 bits to store a physical address; the upper byte is ignored.

An address in the highest 32k bytes of address space is often expressed as a negative number because of the 68000's short addressing mode. With this mode, a signed 16-bit number is sign-extended to 32 bits, specifying an address in either the lowest 32k bytes (positive number) or the highest 32k (negative number). The Pascal system conventionally leaves the upper byte of addresses set to \$FF, so that the decimal integer equivalent of the address of high memory is the value -1 rather than $2^{24}-1$. When writing addresses in hexadecimal, the leading \$FF will be dropped in this text.

In the 9836, the available 16 megabytes are partitioned into areas for ROM, IO interfaces, and RAM. The allowable boundaries of these areas are as follows:



RAM boards are installed from the high end of memory, downward. The Boot ROM checks for the presence of RAM in descending addresses from \$FFFFFF. 9836's have 64k of "floating" RAM mounted on the CPU board. It is called floating RAM because its address is not determined by hardware switches; instead, a special latching circuit causes it to respond to the block of addresses immediately below the lowest-addressed RAM board in the backplane. (If the RAM

board switches are not set contiguously, the floating RAM fills in the first "hole" in the address space as scanned downward from \$FFFFFF.)

The 9836 is built so that accesses to non-present RAM will cause a Bus Error exception. The floating RAM is simply latched to respond to the first address for which a bus error occurs; the Boot ROM is guaranteed to cause such an error during its search for the end of real memory.

It is not possible to change the address of the floating RAM block after power-up. It is possible to have non-contiguous RAM blocks, by incorrectly setting the switches on the memory boards. The Boot ROM will not "find" memory which is not contiguous with address \$FFFFFF, so if you were to set a machine up that way, the stray blocks would have to be accessed by tricks with pointers or address registers.

The Boot ROM resides at address \$000000. There are at least four versions of Boot ROM used in various releases of the 9816, 9826, and 9836 hardware. Pascal 2.0 is designed to run with all of them, but some versions of the Boot ROM are limited as to what devices they can boot from. The sizes of the Boot ROMs range from 16k to 48k. Another section of this manual describes the Boot ROMs in detail, including such information as what device drivers and useful support routines they contain.

Operating system ROMs may be located on 16k byte boundaries beginning with address \$020000 and continuing up to \$3FC000. Such ROMs have special headers which are recognized by the Boot ROM during its search for an operating system. The section of this document describing the Boot ROM tells what ROM headers look like. Accesses to non-present ROM locations do not cause Bus Error exceptions.

The 68000 is built so that when interrupts or exceptions occur, the processor saves (some of) its state and does a kind of forced subroutine call to one of several routines whose addresses are found in locations right above address \$000000. Refer to the CPU manual for the precise correspondence between these "interrupt vector" locations and the various interrupt priority levels and exception conditions. NB: interrupt code runs in Supervisor mode, while programs run in User mode, so different stacks are involved.

9836 family Boot ROMs contain fixed addresses for the exception vectors; they point into high memory right below \$FFFFFF. The exact layout of the area below \$FFFFFF is shown in the discussion of the Boot ROM; it is a mirror reflection of the ROM interrupt vectors themselves, allowing 6 bytes -- enough for a long JMP instruction -- for each vector.

For instance, at address \$00000C (vector 3) the vector content is \$FFFFFF4, so when an Address Error exception takes place, the CPU will call whatever routine is at \$FFFFFF4. The Boot ROM initializes the RAM vector area with JMP instructions leading to an error reporting routine within the Boot ROM itself. Operating systems which subsequently run will change these values as needed.

Below the RAM vectors there is some more memory which is used (and reserved permanently) by the Boot ROM. Below that is some memory which is used during the Boot load but may be used for data after booting.

Version 3.0 of the Boot ROM used more memory than previous versions. It not only takes more space at the upper end of memory, it also may steal a little at the bottom of physical memory. This only happens when booting from certain specialized devices such as the Shared Resource Manager.

To find out how much space was taken at the bottom of memory, examine the integer addressed 20 bytes beyond the location addressed by the four-byte pointer stored in absolute location \$FFED4 (-300). Got that?

```

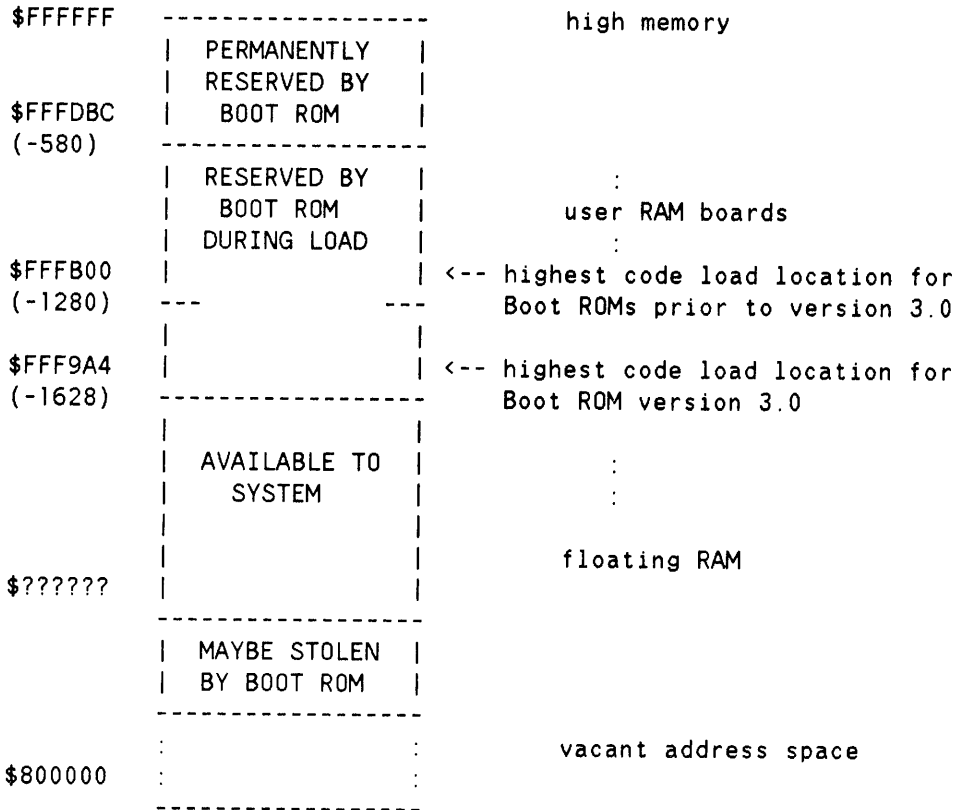
type
  lrec = packed record
    filler: packed array [0..19] of char;
    memused: integer;
  end;
var
  thing [-300]: ^lrec;

  bottombytesstolen := thing^.memused;

```

But you should only do this if the machine has the Version 3.0 Boot ROM. See the chapter on the Boot ROM for how to determine whether to do this. In the memory maps which follow, we will indicate stolen space even though its size may be zero!

So at the point when the Boot ROM is about to find and load a system, the available memory for the system to use is from the near the bottom of physical memory as determined by the RAM board switches and floating RAM, up to a limit determined by the Boot ROM.



If the Boot ROM finds a "soft" system somewhere (in our case Pascal), it now loads that system into RAM. The soft system load is an absolute load; that is, the boot file consists of one or more segments of code which are placed at specific locations in memory -- the particular load addresses are specified in the file itself.

This absolute image is NOT a standard Pascal object code file; it is in a different, much simpler, format. Boot files are generated from linked, relocatable object code by the Librarian's B (Boot)

command. The loader in the Boot ROM is really dumb, so there can be no unsatisfied externals in a boot file. It must be complete and ready to go. It is the responsibility of the programmer generating the boot file to decide where in free RAM the system being loaded must be placed. Usually the boot file consists of a single segment which is placed so that it just snuggles up to address \$FFFAC0.

The Boot Rom runs in 68000 Supervisor mode; most of Pascal runs in User mode. Generally we will use the name "SP" to designate the User mode stack register, and "SSP" for the Supervisor stack register.

The Pascal kernel (the linking loader and minimal other support) is placed in this fashion. Execution begins in **LOADER**, which will use drivers in the Boot ROM to load **INITLIB**. **LOADER** calls a small assembly language entry point (**POWERUP**) containing code for interrupt and trap handling, as well as **TRY-RECOVER** and non-local **GOTO** processing. **POWERUP** performs these actions:

- Sets up a minimal Pascal execution environment consisting of a stack pointer (A7, also called SSP register), stack frame base (A6,also called SF), global variable base pointer (A5), and heap pointer.
- Sets the **TRY-RECOVER** chain and list of open files to empty.
- Sets some (not all!) of the RAM vectors to point to exception handlers within **POWERUP**.

At the moment POWERUP returns to LOADER, the memory map looks like this:

\$FFFFFF	----- RAM VECTORS -----	high memory (correctly initialized)
	RESERVED FOR USE BY BOOT ROM -----	
\$FFF9A4		
	LOADER CODE -----	<--- address of POWERUP code
\$???????		<--- SSP base of supervisor stack (grows downward)
	5000 bytes SUPR. STACK -----	
		<--- base of all global vars; also A6=stack frame pointer
	GLOBAL VARS -----	
		<--- don't know where this is yet
	: : FREE SPACE : : -----	<--- A5 = 32768 less than base of all global vars
\$???????		<--- heap pointer (grows upward) (=lowest useable mem)
	MAYBE STOLEN BY BOOT ROM -----	
		<--- bottom of real RAM

In this structure, if any interrupts occur they will happen "on-top-of" the supervisor stack.

This arrangement may seem a little odd to you. Normally when Pascal code executes, the stack grows downward through free space and the heap grows upward; if they collide, a "stack overflow" error has occurred. Rest assured that the user stack pointer will be moved down below the global variables and all will be well. But while the loader is executing, the stack is in this funny place and it must never be allowed to get so big that it writes over the global variable area.

You may also be wondering why A5 points 32768 bytes below the base of the global variable area. All globals are addressed using the mode "displacement(A5)". Register A5 will never move while Pascal code is running. To allow for full 64k of global space accessible by a 16 bit displacement, the range of displacements used must be -32768..32767; so the base of the global area is exactly 32766(A5).

(This is a departure from the earlier Pascal 1.0 release, in which A5 addresses the beginning -- highest address -- of the global area. The 1.0 system could allow a maximum of only 32k bytes of global area; since the space "above" A5 was occupied by the system stack, only negative displacements from A5 were useable. Unfortunately some of the code in the Boot ROM and the BASIC language ROMs assumes the convention of the 1.0 system, so in Pascal 2.0 ROM code must be accessed through the special interfacing routine ROMCALL.)

Later we will go through a more detailed commentary on the kernel's modular structure. For now it is useful to know that certain kinds of initialization occur which have the effect of consuming some heap space to store system tables and variables including:

- Access method and file suffix tables
- Device driver tables
- The Unitable array

Also, the loader is initialized. This involves finding two pieces of information: how big is the global variable area of the loader itself, and where is the SYSDEFS table.

The loader will allocate space for global variables of new modules, as they are loaded, by extending the limit of the global area downward toward the heap. It must know where to start, ie, how much space is already taken up by the loader's globals?

SYSDEFS is a trick played on the loader. Earlier it was mentioned that as modules are loaded, the loader will keep their symbol tables around so other things can be linked to them later. But the loader itself is not loaded by the loader; it is loaded by the Boot ROM. So we must fake up an area of memory which "looks like" tables built by the loader, to describe the kernel itself. This is called SYSDEFS and is part of the absolute kernel image.

Now we are about to load INITLIB, and the memory map looks like this:

\$FFFFFF	RAM VECTORS		high memory
	RESERVED FOR USE BY BOOT ROM		
\$FFF9A4			
	LOADER CODE	<---	address of POWERUP code
\$???????		<---	base of system stack
	3000 bytes	<---	A6 = current stack frame
	SUP. STACK	<---	SSP = top of system stack (grows downward)
	GLOBAL VARS	<---	base of all global vars
		<---	last global variable
		<---	user stack base will be here
	FREE SPACE	<---	A5 = global base - 32768
		<---	heap pointer
	VARIOUS SYSTEM TABLES		accessed through pointers in global area
\$???????		<---	bottom of useable memory
	MAYBE STOLEN BY BOOT ROM	<---	bottom of ream RAM

The stack frame pointer (A6) got moved into the system stack area as a side effect of the normal Pascal procedure entry mechanism. It always points to the base of the stack frame for the currently executing procedure.

Now the loader loads INITLIB. All the modules are loaded. The code for each module is placed in the system's heap, that is, above the system tables. The globals for each module are added to the system global area, which is growing down toward the heap.

Modules in INITLIB should have no external references which cannot be satisfied by linking either to the kernel itself, or to other modules in INITLIB. This restriction is made because the system has not yet located the system LIBRARY, which could otherwise be used to satisfy external references.

When INITLIB is completely loaded, all the programs it contained (modules with start addresses) are executed in turn. This gives the various subsystems such as IO drivers an opportunity to install their names (addresses) in system tables, to steal heap space, and so forth. Installation code runs not on the system stack, but in its "proper" stack area below the global variable area. This moving of the stack pointer will occur any time a program is executed from now on.

After the running of installation code is finished, the system memory map looks like this:

\$FFFFFF	RAM VECTORS		high memory
	RESERVED FOR USE BY BOOT ROM		
\$FFF9A4			
	LOADER CODE	<---	address of POWERUP code
\$???????		<---	base of supervisor stack (grows downward)
	5000 bytes	<---	A6 = current stack frame
		<---	current SSP
	LOADER GLOBALS	<---	base of all global vars
	INITLIB GLOBALS		
		<---	last global variable
		<---	user stack base will be here
	FREE SPACE	<---	A5 = global base - 32768 (value unchanged)
		<---	heap pointer
	HEAP SPACE STOLEN BY INSTALLATION ROUTINES		such things as workspace required by file system access methods
	INITLIB CODE MODULES		includes loader symbol tables for each module
	VARIOUS SYSTEM TABLES		accessed through pointers in global area
\$???????		<---	bottom of useable memory
	MAYBE STOLEN BY BOOT ROM		

The pattern from here on should be clear. During loading operations, the loader runs its stack in the small "system stack" area. It pushes code onto the heap, and allocates space for module globals

downward. When a program is to run, the user mode stack pointer for the program is set up just below the last global variable.

If a module or program is loaded permanently, the limits of the heap and global area are permanently extended. Running a program which has been permanently loaded is particularly easy since its code and global areas already exist. One need only switch the stack pointer to the user stack area. If the module is to be loaded, executed, then removed to run another program, the heap and global areas can be cut back after the program completes by the amount they were extended.

The maximum allowable global area reaches from 32767(A5) to -32768(A5). System globals are mingled with program globals, and the sum can't exceed 64k bytes.

To complete the Pascal boot process, it remains to load the the Command Interpreter program STARTUP and the IO configuration program TABLE, and to execute them. If STARTUP is found on the boot volume, it is loaded before table; otherwise STARTUP is loaded from the system volume after TABLE executes. This sequence is caused by the last module of installation code in INITLIB, a short program called LAST. The space consumed by TABLE is reclaimed before STARTUP runs. Exactly how TABLE does its job is discussed elsewhere.

It has already been mentioned that STARTUP can be any program. The standard Command Interpreter (CI) supplied with Pascal pulls one last trick. It begins execution in the usual way, but the first thing it does is to switch to the Supervisor Stack register SSP, so it runs in the small stack area above the global variables. This is done because the CI must be able to call the loader to load programs. The loader would be unable to allocate global space for an incoming program if the CI's stack were in the way. So the system stack area was made big enough to safely run not only the kernel but also the Command Interpreter. By the way -- the little routine called LAST at the end of INITLIB also does this, and for the same reasons!

A good way to think about all this is to consider that each module or program which is loaded is bound into (becomes an extension of) the kernel.

This interpretation seems especially appropriate if one considers the capability to permanently load programs or modules using the CI's "P" command. Permanently loaded modules, whether from INITLIB or loaded by the CI, can significantly extend the capabilities of the system. For instance, a new IO driver, or a directory access method of your own invention can easily be added to the system and then accessed freely by other programs. As we will see in discussing the File and IO subsystems, these new capabilities can be made part of the normal access paths of Pascal programs.

One last picture: here is what memory might look like after a number of modules have been permanently loaded and while a program is running.

\$FFFFFF	----- RAM VECTORS -----	high memory
\$FFF9A4	----- RESERVED FOR USE BY BOOT ROM -----	
\$???????	----- LOADER CODE -----	<--- address of POWERUP code
		<--- base of supervisor stack (grows downward)
		<--- base of system/CI stack
	5000 bytes	<--- A6 = current stack frame
		<--- CI's current stack pointer
	----- -----	<--- base of all global vars
	KERNEL GLOBALS -----	
	INITLIB GLOBALS -----	
	CI GLOBALS -----	
	GLOBALS FOR P-LOADED MODULES/PROGRAMS -----	<--- A5 (global base -32768) (value unchanged)
	GLOBALS FOR PROGRAM WHICH IS RUNNING -----	<--- last global variable
	PROGRAM STACK -----	<--- user program's stack base
		<--- A6 (top stack frame)
		<--- SP (top of user stack)
	: FREE SPACE : -----	
	USER HEAP -----	<--- heap pointer
	P-LOADED MODULES -----	either programs or simple modules may be here
	HEAP SPACE STOLEN BY INSTALLATION ROUTINES -----	such things as workspace required by file system access methods
	INITLIB CODE MODULES -----	includes loader symbol tables for each module
\$???????	----- VARIOUS SYSTEM TABLES -----	accessed through pointers in global area
	MAYBE STOLEN BY BOOT ROM -----	<--- bottom of useable memory
		<--- bottom of real RAM

Summary of the Booting Process

The purpose of the booting process is to construct a complete operating environment from the absolute memory image kernel and the contents of the relocatable library INITLIB. When this process is complete, programs can be loaded and executed by the Command Interpreter.

To load a program, its code is put into system heap space (growing the heap upward) and its global area is appended to the system global area (growing downward). The CI and Loader execute out of a special "system stack" area, while a running program bases its stack just below the last global variable.

The system now presents the operator with a simple structure in which subsystems are just programs to be loaded and executed:

Memory resident portions	Loadable subsystems
kernel	
file system	
debugger (optional)	
user-code from INITLIB	
command interpreter	-----> Editor
	Compiler
	Assembler
	File manager
	Librarian
	any executable program

Of course, the Command Interpreter provides a little creature comfort by allowing single-keystroke commands to load the main subsystems, and by providing some automatic flow of control for the process of editing, compiling and running a program. Workfiles are also artifacts of the Command Interpreter.

The Pascal Kernel

The word "kernel" is used advisedly. The Pascal system has no kernel in the closed sense of operating systems such as UNIX, because Pascal has been designed in an open, dynamically extensible way. Pascal boots in a linking loader, which is always resident thereafter. This loader is a sort of "induction rule", by means of which modules of code can be successively added to the system -- while it is running -- to give it more capabilities. As used here the word "kernel" roughly means a reasonable set of useful modules such as the File System, Directory Access Methods and so forth.

The "important" code in the kernel is mostly concerned with two matters: file support and the loading/linking of object code. However, there are a lot of miscellaneous details which complicate the picture. The purpose of this section is to give a good overview, particularly of the file system, so that you can more easily make sense of the code listings themselves.

Refresher on Pascal Modules

If you are quite familiar with Pascal modules, you may wish to skip this section, which describes the relationship of the declared components of a module to entities in its object code form.

A sample module:

```
module Charlie;
import Sue;
export
  const
    lo = 0; hi = 100;
  type
    int = -32768..32767;
    index = lo..hi;
    arrae = array [index] of int;
  var
    head,tail: index;
    list: arrae;
  procedure addtolist (k:int);
  procedure takefromlist (var k:int);
implement
  var
    lastvalue: int;

  procedure hiddenproc;
  begin
    .... body of hiddenproc omitted for clarity
  end;

  procedure addtolist (k:int);
  begin
    ....
  end;

  procedure takefromlist (var k:int);
  begin
    ....
  end;
end {of module charlie};
```

Module Charlie exports the identifiers lo, hi, int, index, arrae, head, tail, list, addtolist and takefromlist. It is declared to be dependent on things imported from module Sue.

When Charlie is compiled, the Compiler will find Sue somewhere and read its export text, so that references to objects of Sue's can be verified syntactically. Similarly, the final object code for Charlie will contain Charlie's export text (with the reference to Sue).

The things declared after the keyword "implement" are said to be "hidden". This means that procedure hiddenproc and variable lastvalue are not visible to an importer of Charlie; only code within the implement part of Charlie itself can access or change the hidden things. The Compiler is responsible for enforcing this secrecy.

In the final object code there will be DEFINED the following load-time symbols, said to be the load-time symbol table for the module. Notice that almost all the original Pascal identifiers, especially names of constants, types and variables, are unknown to the loader.

CHARLIE
CHARLIE_CHARLIE
CHARLIE_ADDTOLIST
CHARLIE_TAKEFROMLIST

CHARLIE is the symbol used to access any global variables of the module. When the loader allocates space for the module's globals, by extending the system global area downward, CHARLIE will be assigned the address of the first even byte above the allocated global area. The variables of the module are below the symbol CHARLIE. To assign the value zero to the int variable called head, we might write in assembly language,

```
MOVE.W #0,CHARLIE-2(A5)
```

The move is word-wide (.W) because the Compiler was smart enough to use a single word to represent a 16 bit subrange. Register A5 always points 32768 bytes below the the top end of all the globals ever allocated in the system, and CHARLIE is given by the loader a value equal to the distance from where A5 points to the high end of Charlie's global area. The offset -2 indicates that head is two bytes below the top end of Charlie's globals.

CHARLIE_CHARLIE is the address of the module initialization body, a subroutine generated automatically by the Compiler. Every compiled module gets one of these. Often it does nothing, but in two circumstances it is vital. If the module contains any file variables hidden in its implement part, they must be initialized to the "closed" state before use by the file system. If the module imports other modules, their initialization bodies must in turn be called.

The whole chain of initializations is started by the main program, which automatically calls the initialization bodies of any modules it imports. They go on to initialize whatever modules they import. Note that initialization bodies are cleverly coded so that only the first call has any effect. This is necessary because a module might be imported several times. This trick is accomplished by taking advantage of the fact that a module's global area is set to all zero's once, when it is loaded.

CHARLIE_ADDTOLIST and CHARLIE_TAKEFROMLIST are exported procedures. These symbols get the module-relative address of the corresponding Pascal procedures. That is, the value of CHARLIE_ADDTOLIST is the distance from the first instruction of the module's code segment to the first instruction of procedure addtolist. No symbol is DEFINED for a hidden procedure.

Modules in the Kernel

The kernel is written as a set of (mostly) Pascal modules. In the usual style of modules, some of them are dependent on (IMPORT) others. The result is a sort of "directed graph" of dependencies. Not all these modules need be in a Pascal system; they are the ones required to give "complete" support to a Pascal program. By removing modules from INITLIB, a rather smaller kernel can be generated. The smallest possible kernel, consisting of only the linking loader, is about 20k bytes in size.

Recall that in this system, it is possible to write assembly language modules which "look like" compiled modules in that they include interface specification -- EXPORT text -- recognizable by the compiler. The module ASM in the list below is such a case, while POWERUP is assembly code which has NO export text. All the other modules listed below are written in Pascal. Actually there are also some very specialized assembly language routines which are not defined as Pascal modules but rather are accessed as EXTERNAL procedures. (These specialized modules are not included in the list below.)

SYSGLOBALS	Declares constants, variables, and types used throughout the system.
ASM	Various high-speed functions such as moving bytes around. Includes the entry point ASM_POWERUP referred to in the boot process, which contains code to handle non-local GOTOs and RECOVER clauses.
INITLOAD	Entry point into system. Calls POWERUP to get dirty work of machine initialization done.
LOADER	Defines directory of a code module, and internal symbol table structures. Manages global and code space. Links and loads object code.
ISR	Sets up and manipulates interrupt service routines.
KBD	Code to handle the keyboard, character set mapping, typeahead buffer, CRT, system clock, battery backup, and powerfail. Actually this is 5 modules linked together for compactness.
MISC	Error messages, directory access method for volumes with no directory, access method for unbuffered transfers, access method for data files (general purpose buffering), access method for serial devices (text). Fills in Access Method, Suffix and Eft tables (explained below). Defines the generalized file catalogue entry type.
MINI	Driver for built-in minifloppies; calls code in Boot ROM.
INITUNITS	Generates the initial IO Unitable setup.
FS	File system functions which are called by the Compiler to implement Pascal file IO. More of this in INITLIB.
SETUPSYS	Calls initialization routines for modules POWERUP, MISC, INITUNITS, LDR, FS.

The dependencies of these modules on each other is shown using the convention that A < B C means Pascal module A directly imports modules B and C; or if A is in assembly language it somehow accesses things exported from B and C.

```

SYSGLOBALS < nothing
POWERUP    < SYSGLOBALS
ASM        < SYSGLOBALS
LOADER     < SYSGLOBALS ASM
ISR        < SYSGLOBALS ASM
MINI       < SYSGLOBALS ASM
BOOTDAMMODULE
           < SYSGLOBALS ASM MINI
INITLOAD   < SYSGLOBALS ASM BOOTDAMMODULE LOADER

```

Everything else also uses SYSGLOBALS and ASM. Moreover,

```
KBD      <  ISR
MISC     <    KBD
FS       <    KBD      MISC
INITUNITS <    KBD MINI      FS
LDR      <    KBD      MISC FS LOADER
SETUPSYS <    KBD      MISC FS LOADER LDR INITUNITS
```

Digression on a Trick

All of these modules provide functionality which can be called from user programs by importing the required modules. This seems like a good moment to explain a subtle point in that regard.

When a module is loaded, the loader keeps its load-time symbol table around. The loader tables can be used to find the value of (address of) exported procedures, global variable areas, and so forth. This was alluded to in the explanation of the booting process, when we noted that the structure called SYSDEFS provides the loader tables for the absolute memory-image kernel. Thus the loader can link a piece of compiled code to things in the kernel.

However, in order to **IMPORT** a module, the Compiler must be able to find that module's interface text in the unlinked object code of the module. The loader doesn't store interface text in memory, just symbol values, because the interface text is only useful during compilation, not during linking or loading. Also it would consume a lot of **RAM**.

The kernel is supplied in a linked, absolute form. So where is its interface specification, that it may be imported? We trick the system by putting the interface specification in modules corresponding to the kernel modules listed above. These modules are dummies; there is no code, since the code by definition is always resident in memory and available to the linking loader. The dummy modules are found in the **INTERFACE** file on the **LIB** disc.

Chapter 7

The File System

Introduction

The purpose of the file system is to provide user programs with a clearly defined set of file operations. These operations must behave uniformly over a variety of device types, directory structures, and file structures. For instance, a program must be able to access or generate a text file properly under any of the following representations:

- An unblocked stream of bytes, eg from the keyboard or going out to a printer.
- A sequence of bytes in a disc file, with ends of lines denoted by <carriage return> characters.
- A file in the WS1.0 text file format, which includes leading blank compression and peculiar blocking characteristics.
- An ASCII file as specified by HP's Logical Interchange Format (LIF) standard, where lines are represented by a 16-bit length field followed by data.

The file system also supports several disc file directory formats, and more can be added by the user without regenerating the kernel. The directory organizations in the 2.0 system are:

- **WS 1.0** Compatible with Pascal 1.0 file system.
- **LIF** HP's Logical Interchange Format for data exchange; supports contiguous files.
- **SRM** (Shared Resource Manager) Remote file service with hierarchical directories and non-contiguous files.
- **Unblocked** For devices without directories (like printers).
- **Boot** Only used during boot process; won't work after.

Finally, the file system isolates the definition of the directory and data transfer operations from the details of the physical driver routines which control operation of peripheral devices.

It was a challenge to unify all these features and at the same time allow flexibility for future extensions such as the addition of new IO device drivers or directory methods without the necessity of regenerating the kernel. The scope and uniformity of the file system is the most important difference between the 1.0 and 2.0 versions of Pascal, and is the reason object code is incompatible between these systems.

Note that in this system there is a sharp distinction between file IO and device IO. File IO is provided by the standard statements of Pascal such as RESET, REWRITE, GET and PUT. Device IO is provided by modules in INITLIB (or IO on the LIB: disc). The reason for this distinction is that there are many disorderly details of the control of physical IO which do not properly belong

in a language definition, aren't interesting to most applications, and vary significantly from one computer family to another. However, the file system uses the physical IO system to actually perform operations to the physical devices.

Representation of File Variables

A File Information Block (FIB) is the data structure which represents a Pascal file variable. It consists of three main parts: the file description, the file window (current record), and the physical buffer. Sometimes the window and physical buffer are not present.

FIB's are Pascal RECORDs -- complicated objects -- whose full description is exported from module SYSGLOBALS. Three particularly important fields of an FIB record are:

- **fkind** The file type.
- **am** The Access Method used by the file.
- **funit** The number of the logical unit on which the file resides.

Access Methods and logical units will be described momentarily.

Files are considered to have a type. There are presently seven recognized types of file, with placeholders for nine more types in the future. The seven filekinds now are:

- **untypedfile** Used for directory entries
- **badfile** Bad blocks on disc
- **codefile** Object code
- **textfile** WS1.0 format text
- **asciifile** HP LIF ASCII strings
- **datafile** Pascal "file of x"
- **sysfile** System boot file

High-Level File Operations

The highest, most unified level of the file system is called File Support (FS). This level consists of the routines called by the Pascal Compiler as it translates program statements. The calls to the FS level are calls to these procedures which are exported from modules FS and MFS ("More FS"):

fbufferref	make sure file window F^ is valid
fblockio	UCSD block read/write
fcloseit	close file
feof	end of file?
feoln	end of text line?
fget	Pascal GET
fgotoxy	position logical cursor
fhopen	open a file
fhpreset	reset file
fmaxpos	where is end of file?
foverprint	reprint same line
fpage	emit formfeed
fposition	what record are we at?
fput	Pascal PUT
fread	read a record
freadbool	read a boolean value
freadchar	read one char
freadenum	read enumerated scalar by name
freadint	read one integer
freadpaoc	read packed array of char from text
freadreal	read real number (** in INITLIB **)
freadstr	read a string
freadln	flush out end-of-line
fseek	position to record randomly
fwrite	write a record
fwritebool	write a boolean value
fwritechar	write one char
fwriteenum	write name of enumerated scalar
fwriteint	write one integer
fwriteln	write end of line
fwritepaoc	write packed array of char to text
fritereal	write real number (** in INITLIB **)
fritestr	write a string
friteword	write a 16-bit integer

Each of these routines requires a FIB as one parameter. See chapter 8 for details concerning these operations.

The Access Methods

The Access Methods are called by File Support to implement buffering or packing of data into (unpacking of data from) the format of physical records on the disc storage medium. For instance, an AM receives the data produced by formatted Pascal write statements to a text file variable and generates the LIF representation of text lines as ASCII strings. Generally speaking, there is an AM for each filekind (type of file).

The things an AM can do are enumerated by a scalar type, `amrequesttype`, declared in module `SYSGLOBALS`. Note that not every AM is expected to be able to do all of these.

```
readbytes
writebytes
flush
writeeol
readtoeol
clearunit
setcursor
getcursor
startread
startwrite
unitstatus
```

These are the components of the scalar type "amrequesttype".

Each FIB has exactly one AM associated with it, in the form of a "procedure variable". (See the System Programming Language Extensions for details on procedure variables. Stated simply, a procedure variable is a variable whose value is the name of a procedure which may be called. A procedure value is the same concept provided by QUOTE in Lisp -- it suppresses evaluation of the procedure.

Use of procedure variables confers a special flexibility on the file system, because their values (names of particular procedures) need not be filled in until run-time. In fact, the procedures can be ones which didn't even exist at the time the kernel was built, as long as they have appropriate parameter lists and supply the required functionality. This is one of the ways modules in `INITLIB` can dramatically extend the capabilities of the kernel.

Formally, an AM is a procedure with the following parameter list:

```
type amtype = procedure (fp:fibp;
                        request:amrequesttype;
                        anyvar buffer>window;
                        bufsize,position:integer);
```

Where an `fibp` is a pointer to an FIB, and a `window` is an array of bytes. There are several AM's supplied with the system; you could add more if you wanted to.

- **unbufferedam** Expects to do a transfer directly to the device, using the Transfer Method for the unit. Used for unblocked devices and for UCSD "untyped file" construct. Find it in module `MISC`.
- **standardam** General purpose buffering, used for Pascal data files (file of x). In `MISC`.
- **textam** UCSD text file format (skip page 0, leading blank compression, nulls at end of page). In `UCSD__am`.

- **asciim** HP LIF ASCII text files (16-bit length plus data for each line). In **ASCIIMODULE**.
- **SRMam** Shared Resource Manager stream-of-bytes structure; similar to UNIX files. In **SRMAM**.
- **serialtextam** Converts the ASCII carriage return character to textfile EOLN conditions for input serial devices such as the keyboard.

Some rules and facts about AM behavior: If a physical buffer is allocated by the Compiler to the **FIB** (which is the case for all files except the UCSD untyped file) then the AM must be able to transfer any number of bytes to or from the buffer starting at any arbitrary memory address (even or odd). The AM also must check for exceeding logical end of file. If the transfer is an output which would exceed the physical end-of-file, the AM should call the **DAM** to try to stretch the file to the required size. If the stretch fails, the AM must indicate an IO error by setting **IORESULT**.

The Unit Table

The "Unitable" [sic] is an array of up to 50 so-called logical units. A logical unit number corresponds to the pound-sign notation used in file names, eg #31: . The purpose of the table is to describe the physical characteristics of each device accessible through the file system. Information in a unit entry includes: (among other things)

- **dam:** Procedure variable naming the Directory Access Method to be used for this unit.
- **tm:** Procedure variable naming the Transfer Method (physical driver) to be used for this unit.
- **sc:** Select code; where to find interface card.
- **ba:** HPIB address or SRM node address.
- **uisinteractive:** Indicates whether user can edit input.
- **uisblkd:** True for discs, false for byte stream devices like printers.
- **uvid:** Name of the volume if known.
- **umediavalid:** Medium has had files opened on it, and has not been changed since.
- **uisfixed:** The medium is not removeable
- **ureportchanges:** If false, suppresses messages when drive door opened. (Filer uses this.)

Types `unitentry` and `unitabletype` are declared in module `SYSGLOBALS`. The actual unit table itself resides in OS heap where it is allocated early in the kernel boot process. It is accessed through a pointer called `unitable`, also in `SYSGLOBALS`.

The Transfer Methods

Transfer methods are also called "low-level access methods" or "drivers"; they are the routines called by AMs and DAMs to do physical input or output. A TM procedure variable is associated not with a FIB but with a particular logical unit (a Unitable entry). The TM uses the information in the unit entry to decide what device to operate on and how to handle the device.

Most TM's ultimately do their work by calling routines available through the Pascal device IO library. It turns out that the types of TM request are described by the same scalar type as the Access Method requests, and a TM procedure has the same parameter list form as an AM procedure.

The various TM's are best located by referencing the `TEA__` procedure bodies in program `CTABLE`.

TMs are only required to be able to transfer to or from a disc starting on sector (256 byte) boundaries. The driver may also require that the buffer memory address start on a word boundary, and that the buffer length be an even number of bytes; some older HP disc drives require this. TM's may round an odd number of bytes up to the next even number.

The driver should check that physical end-of-file (PEOF) is not violated. Drivers for unblocked devices like printers will ignore this.

The driver should set `umediavalid` in the unit entry to false if it detects that the disc drive door has been opened, and it may refuse to read or write to a unit if `umediavalid` is false and `ureportchange` is true.

The Directory Access Methods

The association of an FIB with a physical file is made by a DAM, which encapsulates the organization and basic operations on a mass storage file directory. The DAM requests, listed in the scalar "damrequesttype", are:

- `openvolume`
- `getvolumename, setvolumename`
- `getvolumedate, setvolumedate`
- `changename`
- `purgename`
- `createfile, openfile, closefile, purgefile, stretchit`
- `makedirectory, opendir, closedirectory, duplicatelink, openparentdir, catpasswords, setpasswords, lockfile, unlockfile`
- `crunch`
- `catalog`
- `setunitprefix`

A DAM is a procedure with the following parameter list:

```
type dam = procedure (anyvar f: fib;  
                     unum: unitnum;  
                     request: damrequesttype);
```

Where `unum` is an index into the `Unitable`. Notice that DAMs want an FIB, whereas AMs want a pointer to an FIB. Probably the reasons for this are historic, since passing a pointer by value is the same as passing by reference the thing to which it points.

As with TMs, each logical unit entry has an associated DAM. Any one unit can support only one directory type, which is established by the `TABLE` program during boot-up or whenever `TABLE` is explicitly executed by the user.

How the Access Method is Selected

The Pascal standard procedures RESET, REWRITE, and OPEN are calls at the File Support level, generated by the Compiler. At the time a file is opened, the physical name (title) is examined by file support. First it must be determined what logical unit is being selected. The logical unit is designated by one of these notations:

- ":" or no volume name -- The current "default volume" is used.
- "*" or "*:" -- Shorthand for the system volume.
- #31: -- The pound-sign notation gives unit number directly.
- volname: -- The Unitable must have a volume with the given name.

The funit field of the FIB is set to reflect the unit selected.

If the file already exists, its type (which determines the appropriate AM) will be found in the directory in which it resides. Otherwise the file type and hence the AM must be determined by examining the suffix part of the file name, as follows.

The file name is examined for the presence of a suffix (a period followed by five or fewer characters). The recognized suffixes are:

' .BAD'	A file covering a bad block of disc.
' .TEXT'	UCSD format text file.
' .CODE'	Object code file.
' .ASC'	LIF ASCII text file.
' .SYSTEM'	Boot file.
(no suffix)	Pascal "file of x".

Three variables -- suffixtable, amtable and efttable -- declared in SYSGLOBALS and initialized in MISC, are involved in the AM selection process.

```

type
  filekind =          (* known types *)
    (untypedfile,badfile,codefile,textfile,asciifile,
     datafile,sysfile,
     (* room for expansion *)
     fkind7,fkind8,fkind9,fkind10,fkind11,fkind12,
     fkind13,fkind14,lastfkind);

  suffixtype = string[5];
  amtype = procedure (... access method procedure var type ...)

  amtabletype = array [filekind] of amtype;
  suftabletype = array [filekind] of suffixtype;
  efttabletype = array [filekind] of shortint;

  suftableptrtype = ^suftabletype;
  amtableptrtype = ^amtabletype;
  efttableptrtype = ^efttabletype;

var
  amtable: amtableptrtype;
  suffixtable: suftableptrtype;
  efttable: efttableptrtype;

```

The suffixtable is searched for whatever suffix was stripped off the file name. If a match is found, the index of the matching suffixtable entry is the filekind for the file, otherwise filekind = datafile. The type is stored in the fkind field of the FIB.

If the file is anonymous (the opening operation specified no external name) it is always treated as a data file. Anonymous files declared as TEXT type in the program are given type file of char. The outcome of this is that the FIB is assigned a filekind value, which ultimately specifies the Access Method.

The file opening routine now calls the Directory Access Method designated in the Unitable, passing in the FIB. The DAM looks at the FIB and fkind, and selects the AM as follows:

```

  if not uisblkd then          (*serial device*)
    if not fistextvar then am := tm  (*non-TEXT file*)
      else am := serialtextamhook

  else                          (*blocked device*)
    if not fbuffered then am := amtable^[untypedfile]
      else
        if not fistextvar then am := amtable^[datafile]
          else am := amtable^[filekind];

```

Each DAM gets to make its own choices in selecting AM for a file type; as things happen, all our standard DAM's make the same choices, but that is a fact rather than a regulation. Here is a table summarizing the choices.

	fkind = unblocked -----	blocked -----
file of <type>	tm	amtable^[datafile]
TEXT	serialamtexthook	amtable^[fkind]
file;	tm	amtable^[untypedfile]

The "file;" entry corresponds to the UCSD untyped file, which may only be used for block IO operations.

We have not mentioned the External File Type table. Most file systems can keep a designation of file type in the directory on disc. The eftype array can optionally be used to indicate what this external file type as a short integer. It is not a perfectly general mechanism, since the same file type might require different type designators under different DAM's. The DAM may have to perform a translation if this facility is used.

With this overview, we are now ready to discuss the data structures of the file system in more detail.

("UCSD Pascal" is a trademark of the Regents of the University of California.)

Fields of an FIB

Refer to the declaration of type FIB exported from module SYSGLOBALS. An FIB is a Pascal record having the following fields.

FWINDOW: WINDOWP;

The "window" of a file F is the object pointed at by F^. It is treated by the file system as an array of bytes, big enough to hold exactly one component of the type of the file. The window is sometimes called the "buffer variable" (as distinct from the file's buffer).

FWINDOW does not point into the file's physical buffer; rather, the data is moved between the buffer and the window, whose address doesn't change while the file is open. The reason for this technique is that all physical buffers are 512 or 1024 bytes long, and logical records may be broken across physical record boundaries.

FWINDOW is nil for files declared with no type under the Pascal Compiler's UCSD compatibility mode. Such files can only be used for block IO transfers.

This field is initialized by procedure FINITB in module FS; FINITB is called by code emitted by the Compiler. Note that under certain circumstances FWINDOW may be used in ways unrelated to the above description. Particularly in implementing DAMs, FWINDOW may be used with an untyped FIB to access various types of data involved in handling directory entries. User-level programs never see this.

FLISTPTR: FIBP;

All files which are in stack frames or global data areas (ie anywhere but in the heap) are linked together as they are opened via the FLISTPTR field. The list is used to find and close open files when exiting a program or procedure due to error, non-local GOTO, or normal exit.

FLISTPTR is initialized by code generated by the Compiler.

FRECSIZE: INTEGER;

This is the size of a logical record, that is, sizeof(x) in "file of x". The value is zero for UCSD-compatible untyped files. Note that if FRECSIZE = 0 the Compiler has allocated a FIB of the variant with FBUFFERED = false; no physical buffer and no file window.

Initialized by FINITB in module FS, according to a parameter passed by the Compiler.

FKIND: FILEKIND;

Indicates the type of the file (untypedfile, badfile, codefile, textfile, asciifile, datafile, sysfile, etc). FKIND is initialized when the file is opened; see the detailed discussion above ("How the Access Method for an FIB is Selected").

FISTEXTVAR: BOOLEAN;

Indicates that the file was declared as type TEXT as opposed to "file of char". In some Pascals the two are equivalent, but in HP Pascal implementations only things declared as TEXT may be used with formatted reads and writes.

Initialized by FINITB according to a parameter passed by the Compiler.

FBUFFERED: BOOLEAN;

Indicates whether the 512 byte physical buffer is present in the FIB. It is used by the DAM to help select the correct Access Method. The AM could use FBUFFERED to determine whether the Compiler allocated a physical buffer, however, proper selection of the AM by the DAM usually insures that the buffer is there when it is needed.

Initialized by FINITB according to a parameter passed by the Compiler.

FANONYMOUS: BOOLEAN;

A file is anonymous if it was opened without a physical file name, eg REWRITE(F) as opposed to REWRITE(F,'CHARLIE'). Anonymous files will not be LOCKed when closed, since they have no valid name. The DAM is responsible for generating a random file name if the directory structure can't support nameless temporary files.

Initialized by the FS-level calls FHPRESET and FHPOPEN.

FISNEW: BOOLEAN;

True if the physical file was created at this association of FIB to physical file.

Initialized by the DAM in the createnew operation.

FREADABLE, FWRITEABLE: BOOLEAN;

Initialized, maintained and referenced by the File Support level, based on the particular opening operation OPEN, RESET, APPEND or REWRITE. The Compiler passes the access rights to FOPEN. If not (FREADABLE OR FWRITEABLE) then the file is closed.

FREADMODE, FBUFVALID: BOOLEAN;

In vanilla Pascal, as long as a file is open its window variable must be valid. This causes serious problems for interactive files such as the keyboard, because it means at least one character must be input just to open the file.

HP Pascal solves this problem with so-called "lazy IO", which means that the window isn't made valid until it is referenced by some programmatic operation. The validation of the window is automatic, caused by Compiler-emitted calls to an FS routine called FBUFFERREF. The programmer never sees it, and programs written assuming "eager"IO (buffer always valid) will therefore execute properly.

FREADMODE and FBUFVALID are state variables used to control refilling of the window. They are referenced only at the FS level. The four states and their interpretations are:

FREADMODE	FBUFVALID	STATENAME	----- MEANING -----
false	false	Write	A GET must be done before F^ can be filled with the current component.
false	true	Illegal	TSK !
true	false	Lazy	F^ will be filled if it is referenced
true	true	LookAhead	F^ has already been filled

FEOLN: BOOLEAN;

Indicates end-of-line condition. Either file is at its logical end, or the AM has determined that a complete line has been processed.

FEOLN must always indicate whether the most recently read "character" was actually an EOL marker. This requires special handling by text AMs which don't use a visible character to denote end-of-line.

FEOF: BOOLEAN;

The current file position is past the logical end of file. FEOF is valid only in LookAhead state. Initialized, maintained and referenced by FS level.

FMODIFIED: BOOLEAN;

True if some attribute of the file has changed which will require the DAM to access the directory upon file closure. Usually this means the logical end of file has changed.

Initialized by FS routines to false for an old (existing) file and true for a new file. FMODIFIED is set true by FS or the DAM when physical or logical end of file positions change.

FBUFCHANGED: BOOLEAN;

This flag may be used by the AM in any way it wants. Usually it indicates that the physical buffer has been written into and needs to be flushed out to the disc before the file is closed.

Initialized to false by the FS.

FPOS: INTEGER;

This field serves two purposes: indicating the requested size when creating a new file, and indicating the current byte position in the file once open.

When creating a new file, three cases are distinguished:

- N > 0 Requests BLOCKS*512 bytes, where '[<BLOCKS>]' was appended to the file name.
- N = 0 Means no size was specified. Some DAM's will interpret this as a hint to take the largest space available on the disc.
- N < 0 Means '['*']' was appended to the file name. Some DAM's will take this as a hint to use the second largest available space, or half the largest space.

The DAM will probably ignore the above conventions when opening an existing file, but the FS may request the DAM to "stretch" the file later.

When OPENing or RESETing a file, the value of FPOS is set to zero; when opening for APPEND, FPOS is set to the file's logical end-of-file position. Also affected by SEEK, which does no IO but merely changes FPOS.

The AM must update FPOS after every transfer. When closing a file with the 'CRUNCH' option, the logical end-of-file position recorded permanently in the directory will be the most recent value of FPOS. This can cause truncation of a file.

When the DAM is asked to stretch a file (extend its physical end-of-file), FPOS is used temporarily to indicate what is the desired new physical eof. The DAM should try to allocate at least this much, but in any case it should grab a reasonably large piece.

FLEOF: INTEGER;

Logical end-of-file position. Initialized by the DAM to zero for a new file, or the size of an existing file in bytes. Set by FS to zero on a REWRITE operation; and by AM to the maximum of its initial value and any file positions obtained by writing to the file. Used by the DAM upon file closure to determine the new permanent file size.

FPEOF: INTEGER;

Physical end-of-file position. Initialized and maintained by the DAM to reflect the actual size of the file in bytes. Usually this is the number of bytes allocated to the file on the disc.

The Transfer Method looks at FPEOF to determine whether a transfer is legal.

The Access Method looks at it to determine whether FLEOF can safely be advanced. If the desired FLEOF exceeds FPEOF, the AM must call the DAM to stretch the file. If FPEOF is still too small after calling for a stretch, the AM sets IORESULT to IEOF.

FLASTPOS: INTEGER;

Previous file position. May be used by the AM in any way it wants. Usually indicates the correspondence between the physical buffer and the file. FLASTPOS is initialized to minus one by FS.

FREPTCNT: INTEGER;

A general purpose counter. May be used by the AM in any way it wants. Used to implement blank compression in some text file access methods. Initialized to zero by FS.

AM: AMTYPE;

The procedure variable indicating the Access Method to be used with the file. It is initialized by the DAM, as described in more detail above ("How the Access Method for an FIB Is Selected").

NB: The Command Interpreter changes the AM of the system's standard files INPUT and KEYBOARD to accomplish streaming (interpretation of text file as keyboard input).

FSTARTADDRESS: INTEGER;

Execution address in boot file.

The extension word is a kludge in the definition of Logical Interchange directories; it is an integer associated with each directory entry, which can be used in a way determined by the file type (another 16-bit integer in the directory entry).

The LIF DAM uses the extension word in the following way: if the file type is data, the extension indicated the logical end of file within the allocated physical space for the file. If the file a Boot file type, the extension word is taken to be the start address for the system being loaded by the Boot ROM.

FVID: VID;

A string of up to 16 characters, giving the name of the volume on which the file resides. The file system uses this to choose which Unitable entry, hence which DAM, to use in opening the file. See the description above.

The DAM should verify^{*} that the volume name is correct when the file is opened (that is, the name on the volume label matches the name in the Unitable). After file opening, the DAM can use FVID as it wishes, but usually it is used to verify the volume name on closing the file. This is appropriate since not all HP discs can sense when the door has been opened and the medium changed.

The SRM DAM uses FVID to store the master volume password for the SRM if the user ever has occasion to offer it up.

FTID: TID;

A string of up to 16 characters, giving the name of the file. It is initialized by combined efforts of FS and the DAM. FS strips out volume specifier and size specifier; the DAM removes pathnames and passwords.

FTID is used by the Command Interpreter to identify permanently loaded files; when asked to execute a program, the stripped FTID is compared to those in memory. This means you can't execute from the disc any file whose FTID matches one which has been P-loaded.

PATHID: INTEGER;

Path identification token. May be used by the DAM in any way it pleases. The Shared Resource Manager DAM uses it to identify the directory which is the immediate parent of the current file. (Note that any open file or directory on the SRM is identified by a unique integer. If a given file is opened twice, there will be two distinct integers referring to it. The SRM itself remembers the logical mapping from these integer IDs onto physical files.)

PATHID is initialized to minus one by FS.

FILEID: INTEGER;

File identification token. Initialized by the DAM to whatever is appropriate for the TM. Used by the TM (driver) to locate the physical file associated with the FIB.

For most mass storage drivers, it is the byte offset from the beginning of the volume to the start of the file. In this case, the TM adds FILEID to UNITABLE^[FUNIT].BYTEOFFSET and divides by 256 to compute the disc sector. To access a byte within the file, must also add in POSITION for the offset within the file.

Set to zero for "volume transfer" operations.

FUNIT: UNITNUM;

Unit number (index into the Unitable) for the logical unit on which the file resides. FS sets this up according to the volume name, as described earlier. Knowing the unit number, the FS can select the proper DAM, which then picks the right AM.

Also used by the TM to find the hardware description of the device, for example the interface select code or HPIB address.

FBUSY: BOOLEAN;

Set true by the TM checking a unitstatus request if an overlapped IO operation is in progress.

Those are all the fields which are present in every FIB. The following fields may also be present (FIB is a variant record).

FTITLE: FID;

A string of up to 120 characters. This is the "original" file name, with volume name and size specification removed. It is used by the DAM to extract the file name and other information such as path name and passwords.

This field is invalidated as soon as the file is opened, since the variant is overlaid by FBUFFER.

FBUFFER: PACKED ARRAY [0..511] OF CHAR;

Allocated by the Compiler for all files except UCSD untyped files. If present, as indicated by FBUFFERED, the area may be used in any way the AM wants. Usually it buffers one disc block (2 sectors of 256 bytes) of data, since most drivers can only start reads or writes on sector boundaries.

FEFT: SHORTINT;

This is the external file type. Would normally be set to EFTABLE^[FKIND].

FANONCTR: SHORTINT;

Anonymous file counter. Some DAMs must invent a unique name for temporary, new or anonymous files.

FOPTSTRING: STRING255PTR;

This points to a string containing the optional third parameter to OPEN, REWRITE, APPEND. Be careful how you use this -- the scope in which the actual string was declared could go away on you!

FEOT: EOTPROC;

This is a procedure variable which will be called by the driver (TM) at the end of an overlapped IO transfer. Presently the only interfaces which can specify that the IO transfer be overlapped are the UCSD UNITIO operations. The procedure whose name they store here does nothing. This field is a hook for future use.

FFPW: PASSTYPE;

File password. This is set up by the SRM DAM when parsing file names.

FPURGEOLDLINK: BOOLEAN;

This field is assigned a value by the caller of the SRM DAM with a request to duplicate a file link. If it is true, the file's link into its old directory will be purged at the same time.

FOVERWRITTEN: BOOLEAN;

SRM DAM sets this field up for the OVERWRITEFILE and CREATEFILE requests. It is used to decide what to do when processing the CLOSEFILE request.

FLOCKABLE: BOOLEAN;

Set up by SRM DAM on OPENFILE, CREATEFILE, OVERWRITEFILE requests; it is true if the optional third parameter contained 'LOCKABLE'. The default is false. It is used in conjunction with FLOCKED by routines in module LOCKMODULE as well as by SRM DAM.

FLOCKED: BOOLEAN;

FLOCKED is the state variable which controls a workstation's access to files opened LOCKABLE. No file operations are allowed for a lockable file unless FLOCKED is true (which is also the default value even for non-SRM files).

This field is set by SRM DAM on OPENFILE requests and is changed by calls to LOCK, WAITFORLOCK and UNLOCK in LOCKMODULE. When the file is locked, the FIB's workstation-local copy of all file state information is updated with information from the SRM. When the file is unlocked, the SRM is updated and the physical buffer associated with the FIB is flushed. This mechanism assures that at critical times the SRM's state and the FIB's state are in agreement.

FEXTRA: ARRAY [0..5] OF INTEGER;

Space set aside for future expansion.

FEXTRA2: SHORTINT;

Space set aside for future expansion.

FB0, FB1, FB2, FB3: BOOLEAN;

Space set aside for future expansion.


```

    unitabletype = array [unitnum] of unitentry;
    unitableptr = ^unitabletype;

var
    unitable: unitableptr;

```

When we speak of the "nth" unit entry, we really mean "unitable^[n]". The table is allocated out of system heap very early in kernel execution, by module INITUNITS which is in INITLIB. NB: unitable^[0] is used to hold the DAM procedure which will be associated with RAM volumes.

The Fields of a Unit Entry

DAM: DAMTYPE;

This is the procedure variable specifying the Directory Access Method for the volume or device. Initialized during the boot process, usually by execution of the TABLE configuration program just before the Command Interpreter starts.

TM: AMTYPE;

A procedure variable specifying the TM (driver) for the physical device accessed through this unit. Initialized during the boot process, usually by execution of TABLE.

SC: BYTE;

The "select code" for the device interface card. All IO is memory-mapped. Knowing the select code, one can calculate the address in memory of the interface circuitry.

There are 64 possible interface card address areas, each a block of 65k bytes, allocated above \$400000 in the 68000's address space. The calculation of an interface address as a function of select code is not straightforward for reasons having to do with compatibility with previous generations of desktop machines (the 9825, 9835 and 9845). This matter is discussed under Device IO.

I have been asked (too often) how we decide what compatibility to try to preserve and what we are willing to discard. The only honest answer is that compatibility decisions are made in an evolutionary process. Each decision is constrained by previous ones, and it is hard to create an elegant balance among history, innovation and progress. Decisions can be justified instantaneously, but the overall result may be baffling.

BA: BYTE;

"Bus address". Intended to record the HPIB address of the device. Could be used for other purposes by non-HPIB drivers.

This should tell you that several Unitable entries may have the same interface select code, while differing in the value of BA.

DU: BYTE;

"Disc Unit". Selects one disc unit among several being managed by one controller addressed through one select code. For instance, in a 9836 the right minifloppy drive is unit zero and the left is unit one.

This should also tell you that several Unitable entries may have the same select code and HPIB address. The 9134 micro- Winchester disc, for example, looks like four separate units all having the same select code and bus address; they are distinguished only by the value of DU.

DV: BYTE;

"Disc volume". In the new Command-Set '80 family of disc drives, the protocol allows still further partitioning of a particular disc unit into "volumes". These are NOT the same as Pascal volumes !! For such discs, in the future DV will specify the disc volume of interest.

Initialized by TABLE.

BYTEOFFSET: INTEGER;

Gives the byte offset from the start of the disc medium to the start of the volume. This is mainly intended to allow creation of multiple directories on a single physical volume, and typically one finds a volume directory at the start of the volume. In this usage, a volume is a single contiguous area of disc, in which reside a directory and files; the files are accessed relative to the start of volume, rather than start of disc.

A different mechanism is used for hierarchical directories. In fact, the Shared Resource Manager keeps track itself of directory locations; a user "finds" them by name instead of by address.

BYTEOFFSET is initialized by the TABLE program.

DEVID: INTEGER;

"Device ID". This is a misnomer. It is a driver-dependent field currently used in two ways. For CS-80 disc drivers DEVID contains the actual product number of the device, as returned by the "describe" disc command. For local printers, contains the printer byte timeout in milliseconds, as specified by the option in CTABLE.

UVID: VID;

The unit's volume id, a string of up to 16 characters. If the unit is a blocked mass storage (disc) device, UVID gives the name read from the physical volume label; it will change if the disc medium is changed. If the unit is a byte stream device (printer, CRT, keyboard, etc) which has no volume label, the UVID is put in by TABLE and never changes. Thus PRINTER is the UVID for the system print device.

DVRTEMP: INTEGER;

Most transfer methods need some working space to maintain state information. This state information must be maintained with the unit entry rather than the driver module, since the same driver may service several units.

DVRTEMP is a general-purpose variable with which drivers can work their will. For CS-80 and Amigo drivers: while busy, DVRTEMP points to the background temporary space in use. While not busy, contains the IORESULT of the last operation if performed in overlapped mode. For local printers, this field contains the most recent character output, so it can be sent out again if a printer timeout occurs.

LETTER: CHAR;

An archaeological oddity. For a while, within HP there was a plan to identify each type of disc device by a letter, eg "F" for a 9885 floppy disc. Somewhere the plan got lost, because various divisions do their own thing now; anyway, there are too many kinds of disc.

The 9836 tries to follow and extend the same letters used in the older 9835/9845 computers. This letter is in fact examined by the device drivers. For instance, the same driver runs all the Amigo discs; it tells them apart by their distinguishing letters in the unit entry.

LETTER	DISC SELECTED
F	9885 8" single-sided floppy
G	Shared Resource Manager
H	9895 8" double-sided floppy and 9134 micro-Winchester drive
J	any printer
K	streaming backup tape in CS-80 drives
M	internal 5.25" minifloppy
N	8290x family 5.25" minifloppy and 5.25" minifloppy packaged with 9135 Winchester and 3.25" microfloppy
Q	CS-80 family mass storage devices
R	RAM (memory-resident) volume
U	9134A or 9135A micro-Winchester disc (single volume 5 megabyte version)
V	9134B or 9135B micro-Winchester disc (reserved for 10 megabyte version)
W	9134C or 9135C micro-Winchester disc (reserved for 15 megabyte version)
#255	no device flag

There are several important uses for the drive letter:

1. MEDIAINIT uses it to know which medium initialization routine to reference, since there is no TM request to initialize disc media.
2. When the same driver is used to support more than one version of device, the letter tells the driver what to do.
3. A letter is returned by CTABLE's scanning procedures to indicate device type during the boot-up process.
4. A letter is returned by CTABLE's get_bootdevice_parms routine to indicate device parameters.
5. The letter R is noted by CTABLE to avoid overwriting existing RAM volumes. This is relevant if CTABLE is executed by a user command after the system is "up".

OFFLINE: BOOLEAN;

Indicates a blocked device which is absent or malfunctioning; used by drivers to avoid time-consuming attempts at IO -- particularly attempts to find volume directories -- on down devices. This field is (and probably should be) ignored by drivers for byte-stream devices, because an unblocked device doesn't have to be accessible to determine its volume name.

OFFLINE is initialized to false at TABLE execution and by the I (Initialize) command invoked from the main menu. In both instances, a unitclear is performed on all 50 units, and OFFLINE is set true for each blocked unit returning a non-zero IORESULT.

UISINTERACTIVE: BOOLEAN;

Indicates an input device which echoes its data, such as the standard keyboard/CRT or a terminal. This field is usually initialized by TABLE at boot time.

It is referenced by KILLCHAR in module FS. (KILLCHAR assists in editing input data.)

It is referenced by STREAMING in the Command Interpreter to decide whether to "pseudo-echo" the stream file to the screen.

It is referenced by FEOF to decide whether to read 1 character ahead. For instance, unit #1 (CONSOLE: echoing standard input file INPUT) has uisinteractive true, while unit #2 (SYSTEM: non-echoing standard input file KEYBOARD) doesn't. Thus the predicate EOF(KEYBOARD) will force a one-character LookAhead while EOF(INPUT) won't.

UMEDIAVALID: BOOLEAN;

Indicates that open files on the medium in this unit are still valid. The TM (driver) will refuse to read or write to a unit if this flag is false and UREPORTCHANGE is true.

Initialized to false at bootup. Set false by the Command Interpreter whenever recovering from a fatal user program error, and by the I (Initialize) command from the main menu.

Set to false by the DAM or TM whenever there is reason to believe that a removeable medium has been changed (door open bit). Set to true by the DAM whenever it successfully opens or creates any file on the current directory.

If this flag is false when a DAM successfully opens or creates a file, the DAM must find and destroy any temporary (improperly closed) files on the volume. This operation may be thought of as cleaning up the directory of a disc which was removed from the drive while file operations were active.

UISFIXED: BOOLEAN;

If false, the medium is removeable and the driver probably ought to pay attention to whether the disc drive door has been opened. (The driver will generate an ESCAPE(-10) with IORESULT set to ZMEDIUMCHANGED (50).) This is used by the Filer to avoid silly messages instructing the operator to swap discs when the medium is not removeable.

UREPORTCHANGES: BOOLEAN;

If false, the driver ignores UMEDIAVALID. This is used by the Filer to avoid error messages in file copying sequences which require discs to be swapped in the same drive. It may also be used by DAMs to suppress error reports in some circumstances.

UUPPERCASE: BOOLEAN;

Some directory methods want volume names to have no lower- case letters. This flag tells the file specifier scanning routines what to do.

PAD: 0..1;

This bit is presently unused.

UISBLKD: BOOLEAN;

This variant tag field indicates that the device probably has the following characteristics:

- A directory.
- Randomly accessible.
- Can only read or write starting at sector (256 byte) boundaries.

Not all these characteristics need to hold perfectly. For example, no directory may have yet been created on a blocked device. Some devices, such as the streaming backup tape in a 7908 disc drive are only "pseudo-random", and there may be a severe performance or reliability penalty for using such a device as if it were a disc. Likewise, the streaming backup tape works in 1024 byte blocks, so the TM must simulate the behavior of smaller blocks.

UISBLKD is initialized at bootup, usually by TABLE.

UMAXBYTES: INTEGER;

The size in bytes of the volume. If there is only one volume on the disc drive, UMAXBYTES will be the same as the medium size; if there are multiple volumes, each is sized separately.

For most units, this field is constant, having been set up by TABLE. However, for devices supporting removeable media of differing sizes, life is more complex.

- 9885s and 9895s. The 9895 supports both double and single-sided discs. Double-sided discs are always expected to have 150 useable tracks. Single-sided discs are a mess. Depending upon the initializing host computer and the condition of the medium (spared tracks), a single-sided disc may have 61 to 67 tracks, or 73 tracks!

For this reason, UMAXBYTES is normally NOT referenced directly; instead the integer function UEOVBYTES is called passing the unit number. For everything except 9885s and 9895s UMAXBYTES is returned. For 9885s and 9895s, UEOVBYTES actually attempts to access the tracks at the end of the medium to determine exact size. For these two drive types, UMAXBYTES contains the maximum possible medium size in bytes.

- For the streaming backup tape in CS-80 disc drives, attempting to access the device through the file system is VERY inefficient and not recommended except for backup operations. However, at unitclear and medium-change times the CS-80 driver does put the correct value (17 Mbytes or 67 Mbytes) into UMAXBYTES.

Chapter 8

File Support

Introduction

This chapter covers the File Support calls issued by a Compiler to perform file operations. A simple example program is presented, and the calls it issues are discussed. The purpose is to give a better understanding of how program IO actually takes place.

The program creates a file of integers, locks the file, re-opens it for direct access, and reads it in the order opposite to the way it was written using the Pascal READDIR standard procedure. Finally the file is purged.

What follows is a listing and disassembly of the program. After the disassembly is a commentary on the code emitted by the Compiler, which is exactly what one should write to call the File Support level routines from an assembly language program or any other environment.

Since the program was compiled with \$DEBUG ON\$, there is a TRAP #0 instruction followed by a 16-bit line number before the first instruction of each Pascal line.

```
1:D      0 $debug on$ (*Show line numbers*)
2:S
3:D      0 program filedemo (output);
4:D      1 type
5:D      1   ifile = file of integer;
6:D      1 var
7:D     -666 1   f: ifile;
8:D     -678 1   i,j,k: integer;
9:C      1 begin
10:S
11*C     1   rewrite(f,'INTFILE');
12*C     1   for i := 1 to 100 do
13*C     2     write(f,(101-i));
14*C     1   close(f,'LOCK');
15:S
16*C     1   open(f,'INTFILE');
17*C     1   for i := 100 downto 1 do
18:C     2     begin
19*C     2       readdir(f,i,k);
20*C     2       writeln(output,'Record #',i:3,' = ',k:3);
21:C     2     end;
22:S
23*C     1   close(f,'PURGE');
24*C     1 end.
```

No errors.

The Librarian provided the following disassembly.

```
MODULE   FILEDEMO   Created  9-Aug-82
NOTICE:  (none)
  Produced by Pascal Compiler of 26-Jul-82
  Revision number 2
  Directory size   180 bytes
  Module size     2560 bytes
  Execution address Rbase+14
  Code base       0      Size      474 bytes
  Global base     0      Size      682 bytes
  EXT  block  4      Size      136 bytes
  DEF  block  2      Size      62 bytes
  No EXPORT text
  There are      1 TEXT records
```

DEFINE SOURCE of 'FILEDEMO':

DEF table of 'FILEDEMO':

```
FILEDEMO           Gbase
FILEDEMO_FILEDEMO Rbase+14
FILEDEMO__BASE     Rbase
```

EXT table of 'FILEDEMO':

```
FS_FCLOSEIT
FS_FHPOPEN
FS_FINITB
FS_FREAD
FS_FSEEK
FS_FWRITE
FS_FWRITEINT
FS_FWRITELN
FS_FWRITEPAOC
SYSGLOBALS
```

```
TEXT RECORD #      1 of 'FILEDEMO':
  TEXT start block  1      Size      474 bytes
  REF  start block  3      Size      156 bytes
  LOAD address     Rbase
```

```
0 000B          dc.w 11          or dc.b 0,11        or dc.b ' '
2 0946          dc.w 2374         or dc.b 9,70        or dc.b ' F'
4 494C          dc.w 18764        or dc.b 73,76       or dc.b 'IL'
6 4544          dc.w 17732        or dc.b 69,68       or dc.b 'ED'
8 454D          dc.w 17741        or dc.b 69,77       or dc.b 'EM'
10 4F20         dc.w 20256        or dc.b 79,32       or dc.b '0 '
12 0A00         dc.w 2560         or dc.b 10,0        or dc.b ' '
```

```

----- FILEDEMO_FILEDEMO
14 4E56 0000      link a6,#0
18 486D FD66      pea Gbase-666(a5)
22 486D FFFC      pea Gbase-4(a5)
26 2F3C 0000      move.l #4,-(sp)
    0004
32 4EB9 0000      jsr FS_FINITB
    0000
38 41ED FD66      lea Gbase-666(a5),a0
42 216D FFFA      move.l SYSGLOBALS-6(a5),4(a0)
    0004
48 2B48 FFFA      move.l a0,SYSGLOBALS-6(a5)

52 4E40 000B      trap #0,#11          COMPILED LINE NUMBER 11
56 486D FD66      pea Gbase-666(a5)
60 3F3C 0001      move.w #1,-(sp)
64 487A 0188      pea Rbase+458
68 487A 0172      pea Rbase+440
72 4EB9 0000      jsr FS_FHPOPEN
    0000
78 4AAD FFEA      tst.l SYSGLOBALS-22(a5)
82 6702          beq.s Rbase+86
84 4E43          trap #3
86 4E40 000C      trap #0,#12          COMPILED LINE NUMBER 12
90 42AD FD5A      clr.l Gbase-678(a5)
94 52AD FD5A      addq.l #1,Gbase-678(a5)
98 4E40 000D      trap #0,#13          COMPILED LINE NUMBER 13
102 486D FD66     pea Gbase-666(a5)
106 7065          moveq #101,d0
108 90AD FD5A     sub.l Gbase-678(a5),d0
112 4E76          trapv
114 2B40 FD56     move.l d0,Gbase-682(a5)
118 486D FD56     pea Gbase-682(a5)
122 4EB9 0000     jsr FS_FWRITE
    0000
128 4AAD FFEA     tst.l SYSGLOBALS-22(a5)
132 6702          beq.s Rbase+136
134 4E43          trap #3
136 0CAD 0000     cmpi.l #100,Gbase-678(a5)
    0064 FD5A
144 6DCC          blt.s Rbase+94
146 4E40 000E     trap #0,#14          COMPILED LINE NUMBER 14
150 486D FD66     pea Gbase-666(a5)
154 487A 0122     pea Rbase+446
158 4EB9 0000     jsr FS_FCLOSEIT
    0000
164 4AAD FFEA     tst.l SYSGLOBALS-22(a5)
168 6702          beq.s Rbase+172
170 4E43          trap #3

```

```

172 4E40 0010      trap #0,#16          COMPILED LINE NUMBER 16
176 486D FD66      pea Gbase-666(a5)
180 3F3C 0002      move.w #2,-(sp)
184 487A 0110      pea Rbase+458
188 487A 00FA      pea Rbase+440
192 4EB9 0000      jsr FS_FHPOPEN
    0000
198 4AAD FFEA      tst.l SYSGLOBALS-22(a5)
202 6702          beq.s Rbase+206
204 4E43          trap #3
206 4E40 0011      trap #0,#17          COMPILED LINE NUMBER 17
210 2B7C 0000      move.l #101,Gbase-678(a5)
    0065 FD5A
218 53AD FD5A      subq.l #1,Gbase-678(a5)
222 4E40 0013      trap #0,#19          COMPILED LINE NUMBER 19
226 486D FD66      pea Gbase-666(a5)
230 2F17          move.l (sp),-(sp)
232 2F2D FD5A      move.l Gbase-678(a5),-(sp)
236 4EB9 0000      jsr FS_FSEEK
    0000
242 4AAD FFEA      tst.l SYSGLOBALS-22(a5)
246 6702          beq.s Rbase+250
248 4E43          trap #3
250 486D FD62      pea Gbase-670(a5)
254 4EB9 0000      jsr FS_FREAD
    0000
260 4AAD FFEA      tst.l SYSGLOBALS-22(a5)
264 6702          beq.s Rbase+268
266 4E43          trap #3
268 4E40 0014      trap #0,#20          COMPILED LINE NUMBER 20
272 2F2D FFA6      move.l SYSGLOBALS-90(a5),-(sp)
276 2F17          move.l (sp),-(sp)
278 487A 00BA      pea Rbase+466
282 3F3C 0008      move.w #8,-(sp)
286 3F3C FFFF      move.w #-1,-(sp)
290 4EB9 0000      jsr FS_FWRITEPA0C
    0000
296 4AAD FFEA      tst.l SYSGLOBALS-22(a5)
300 6702          beq.s Rbase+304
302 4E43          trap #3
304 2F17          move.l (sp),-(sp)
306 2F2D FD5A      move.l Gbase-678(a5),-(sp)
310 3F3C 0003      move.w #3,-(sp)
314 4EB9 0000      jsr FS_FWRITEINT
    0000
320 4AAD FFEA      tst.l SYSGLOBALS-22(a5)
324 6702          beq.s Rbase+328

```

```

326 4E43          trap #3
328 2F17          move.l (sp),-(sp)
330 487A 006E     pea Rbase+442
334 3F3C 0003     move.w #3,-(sp)
338 3F3C FFFF     move.w #-1,-(sp)
342 4EB9 0000     jsr FS_FWRITEPAOC
      0000
348 4AAD FFEA     tst.l SYSGLOBALS-22(a5)
352 6702          beq.s Rbase+356
354 4E43          trap #3
356 2F17          move.l (sp),-(sp)
358 2F2D FD62     move.l Gbase-670(a5),-(sp)
362 3F3C 0003     move.w #3,-(sp)
366 4EB9 0000     jsr FS_FWRITEINT
      0000
372 4AAD FFEA     tst.l SYSGLOBALS-22(a5)
376 6702          beq.s Rbase+380
378 4E43          trap #3
380 4EB9 0000     jsr FS_FWRITELN
      0000
386 4AAD FFEA     tst.l SYSGLOBALS-22(a5)
390 6702          beq.s Rbase+394
392 4E43          trap #3
394 0CAD 0000     cmpi.l #1,Gbase-678(a5)
      0001 FD5A
402 6E00 FF46     bgt Rbase+218
406 4E40 0017     trap #0,#23          COMPILED LINE NUMBER 23
410 486D FD66     pea Gbase-666(a5)
414 487A 0024     pea Rbase+452
418 4EB9 0000     jsr FS_FCLOSEIT
      0000
424 4AAD FFEA     tst.l SYSGLOBALS-22(a5)
428 6702          beq.s Rbase+432
430 4E43          trap #3
432 4E40 0018     trap #0,#24          COMPILED LINE NUMBER 24
436 4E5E          unlk a6
438 4E75          rts

```

440	0000	dc.w 0	or dc.b 0,0	or dc.b ' '
442	203D	dc.w 8253	or dc.b 32,61	or dc.b '='
444	2000	dc.w 8192	or dc.b 32,0	or dc.b ' '
446	044C	dc.w 1100	or dc.b 4,76	or dc.b 'L'
448	4F43	dc.w 20291	or dc.b 79,67	or dc.b 'OC'
450	4B00	dc.w 19200	or dc.b 75,0	or dc.b 'K'
452	0550	dc.w 1360	or dc.b 5,80	or dc.b 'P'
454	5552	dc.w 21842	or dc.b 85,82	or dc.b 'UR'
456	4745	dc.w 18245	or dc.b 71,69	or dc.b 'GE'
458	0749	dc.w 1865	or dc.b 7,73	or dc.b 'I'
460	4E54	dc.w 20052	or dc.b 78,84	or dc.b 'NT'
462	4649	dc.w 17993	or dc.b 70,73	or dc.b 'FI'
464	4C45	dc.w 19525	or dc.b 76,69	or dc.b 'LE'
466	5265	dc.w 21093	or dc.b 82,101	or dc.b 'Re'
468	636F	dc.w 25455	or dc.b 99,111	or dc.b 'co'
470	7264	dc.w 29284	or dc.b 114,100	or dc.b 'rd'
472	2023	dc.w 8227	or dc.b 32,35	or dc.b '#'

In the following commentary, the notation [nn] refers to Pascal source line number nn; the notation @nn refers to byte offset nn from the beginning of the relocatable text segment. (Byte offsets are the left-hand column of numbers in the disassembly.) The symbol "Gbase" is the relocated base of the global variable area for this program; "Rbase" is the relocated base address where the program's code is ultimately loaded.

@18

Before the first line of user code, the Compiler emits a call to FS__FINITB, which initializes the FIB properly. This must take place before any other operations using the FIB. The file has been allocated 666 bytes of global area as follows: 4 bytes at Gbase-4 for the window variable (size of an integer, which is the file type); 662 bytes for the FIB itself, including a 512-byte physical buffer at the end of the FIB.

Since global areas "grow downward" but variable fields "grow upward", Gbase-666(A5) is the address of the first byte of the FIB while Gbase-4(A5) is the address of the file window variable. The call pushes the address of the FIB, the address of the window, and the size of a record, then calls FS__FINITB.

Then (@38) the FIB is pushed onto a linked list (a stack) of active files. This will enable the system to find and close any open files if the program aborts; it is an optional but highly recommended step. The pointer to the head of the file chain is at SYSGLOBALS-6(A5); it now points to our FIB, and the second field of the FIB, FLISTPTR, is set to point to the next item in the chain.

```

1:D      0 $debug on$ (*Show line numbers*)
2:S
3:D      0 program filedemo (output);
4:D      1 type
5:D      1   ifile = file of integer;
6:D      1 var
7:D     -666 1   f: ifile;
8:D     -678 1   i,j,k: integer;
9:C      1 begin
10:S
11*C     1   rewrite(f, 'INTFILE');
12*C     1   for i := 1 to 100 do
13*C     2     write(f, (101-i) );
14*C     1   close(f, 'LOCK');
15:S
16*C     1   open(f, 'INTFILE');
17*C     1   for i := 100 downto 1 do
18:C     2     begin
19*C     2       readdr(f,i,k);
20*C     2       writeln(output, 'Record #', i:3, ' = ', k:3);
21:C     2     end;
22:S
23*C     1   close(f, 'PURGE');
24*C     1 end.

```

No errors.

[11] @56

This is the call to **REWRITE**, which opens the file for output. The address of the **FIB** is pushed, then a literal value one (1) indicating write-only access, then the address of the string containing the file name, then the address of a null string corresponding to the absent optional 3rd parameter of the **REWRITE** statement. The routine called is **FS_HPOPEN**, which performs all the legal file opening operations.

There are four types of access, exported from module **FS**:

```
type faccess = (readonly, writeonly, readwrite, append);
```

As with all Pascal enumerated scalars, the ordinal values corresponding to these types are 0, 1, 2,

Note that the representation of a string has a leading byte telling the length; length = 0 is perfectly legal.

@78

The Compiler checks the system variable **IORESULT**, because the program was compiled with the default **\$IOCHECKS ON\$**. The **IORESULT** variable is found at **SYSGLOBALS-22(A5)**. If it is zero, the operation was successful; otherwise a **TRAP #3** is executed.

[13] @102

To write the value (101-i) the Compiler emits:

- Push address of FIB.
- Compute (101-i) and store in a variable. The variable is global Gbase-678(A5) because this is the main program; in a procedure some local cell would have been used.
- Push address of local variable.
- Call FS__FWRITE.

FWRITE only needs the address of the value to be written; the size of the component was stored in the FIB by the call to FINITB. After the write, IORESULT is checked again.

[14] @146

Close the file with LOCK. The sequence is:

- Push address of FIB.
- Push address of string 'LOCK' telling what to do.
- Call FS__FCLOSEIT.

[16] @172

Open the file for direct access by the Pascal standard procedure OPEN. This translates into another call on FS__HPOPEN:

- Push FIB address.
- Push literal 2, indicating faccess = readwrite.
- Push address of string containing file name.
- Call FS__HPOPEN.

[19] @226

The standard procedure READDIR is translated by the Compiler into a SEEK followed by a READ. The original call was READDIR(F,I,K) meaning read the value of K from the Ith component of file F:

- Push FIB address for call to READ.
- Push another copy of it for call to SEEK.
- Push the value of the record number (value of I).
- Call FS__FSEEK.

- (Optionally) test IORESULT.
- Push address of variable K which will be read.
- Call FS__FREAD.
- Check IORESULT.

[20] @268-402

These calls are generated by the Pascal WRITELN to standard file OUTPUT. Output is a file like any other file, which is to say it has a physical buffer and a window variable. However, the Compiler happens to know that there is a pointer to the FIB for OUTPUT at address SYSGLOBALS-90(A5); the value of this pointer is the address of the FIB.

The single write statement will translate into a sequence of calls on the appropriate output editing routines to format the data. The FIB pointer is pushed once (@272) and then duplicated on top of the stack as needed for each FS call which will be emitted.

The general form of argument list for textfile input and output routines is: FIB address, value or address of object being read/written, one or two integers for formatting field width specification, and a call to the appropriate routine. For instance, to write a quoted literal the Compiler generates:

- Push FIB address.
- Push address of packed array of characters stored in the constant pool (some Rbase-relative value).
- Push the length of the packed array of characters.
- Push the desired field width (-1 means use actual length of the array).
- Call FS__WRITEPAOC (Write-Packed-Array-of-Char)

[23] @406

Closing the file with 'PURGE' is just like any other closing operation; a string is passed to indicate the disposition.

Only one aspect of file handling was not demonstrated by this example, which is the removal of the FIB from the chain of active FIB's. For global files this is not necessary, since the chain is marked empty just before a program starts running. The Compiler will emit code at block exit to remove from the chain any files whose scope is local to some procedure block. The routine called by the Compiler for this purpose is ASM__CLOSEFILES, which is found in the assembly language module POWERUP.

There is also an automatic removal process which occurs for non-local GOTOs and during TRY-RECOVER processing if it deletes a procedure frame from the stack. ASM__CLOSEFILES is again used.

Files allocated in the heap are not automatically closed, but they are closed if the space in which they reside is DISPOSEd.

Error Reporting by the File IO Subsystems

There is a single, simple error reporting mechanism used for errors of file IO. Exported from module SYSGLOBALS is a variable called SYSIORESULT, also accessible as the "system function" IORESULT, which is translated by the Compiler into a direct access to the system variable.

All Pascal statements which translate into FS-level calls (such as READ, WRITE, GET, RESET) are handled specially by the Compiler. When the directive \$IOCHECKS ON\$ is active (which is the default case), code is emitted after every FS-level call to verify that IORESULT is zero. If it is nonzero, an automatic call ESCAPE(-10) is generated, which will be reported as an IO error unless it is trapped by TRY-RECOVER somewhere.

The FS, AMs, DAMs, and TMs are all compiled with \$IOCHECKS OFF\$, and must explicitly check for ioresults where appropriate. If you write an AM, a DAM, or a TM you will need to do likewise.

The values of IORESULT are also exported from SYSGLOBALS. They are repeated here for easy reference. Note that they are divided into two mutually exclusive groups: those beginning with 'z' are reserved for low-level drivers, while those beginning with 'i' are reserved for the higher level routines.

0	inoerror	no error occurred on last IO call.
1	zbadblock	CRC error (failed disc sector after retries).
2	ibadunit	illegal unit number (1..50 are legal).
3	zbadmode	TM doesn't know how to do requested transfer.
4	ztimeout	device not responding.
5	ilostunit	volume name on unit doesn't match Unitable.
6	ilostfile	file was purged while open to another FIB.
7	ibadttitle	illegal syntax for file name in this DAM.
8	inoroom	no file space; can't create or extend file.
9	inounit	named volume not found.
10	inofile	named file not found.
11	idupfile	DAM doesn't allow two files with same name.
12	inotclosed	tried to open an already open file.
13	inotopen	tried to close an already closed file.
14	ibadformat	bad input data to formatted numeric read.
15	znosuchblk	attempt to read or write past volume limits.
16	znodevice	device offline.
17	zinitfail	initialization of medium failed.
18	zprotected	the medium is write-protected.
19	zstrangei	unexpected interrupt.
20	zbadhardware	hardware or medium failed.
21	zcatchall	ouch -- some kind of driver problem.
22	zbaddma	DMA interface card failed.
23	inotvalidsize	specified file size incompatible w/file type.
24	inotreadable	file not opened for reading.
25	inotwriteable	file not opened for writing.
26	inotdirect	file not opened for random access.
27	idirfull	directory is full.
28	istrovfl	string bound violation in STRWRITE/STRREAD.
29	ibadclos	bad file disposition parameter to CLOSE.
30	ieof	tried to read past logical end of file.
31	zuninitialized	tried to use an uninitialized disc medium.
32	znoblock	block not found on medium (usually bad disc).
33	znotready	device not ready.
34	znomedium	no storage medium mounted in drive.
35	inodirectory	no directory/not readable by this DAM.
36	ibadfiletype	file type designator not recognized by AM.
37	ibadvalue	some parameter illegal or out of range.
38	icantstretch	file cannot be extended.
39	ibadrequest	DAM or AM can't perform requested service.
40	inotlockable	file not opened "lockable".
41	ifilelocked	file already in locked state.
42	ifileunlocked	attempted IO on lockable but unlocked file.
43	idirnotempty	tried to remove non-empty SRM directory.
44	itoomanyopen	SRM: too many open files on device.
45	inoaccess	password required for this access.
46	ibadpass	invalid password offered to SRM.
47	ifilenotdir	the file is not a directory.
48	inotondir	operation not allowed/supported on directory
49	ineedtempdir	couldn't create /WORKSTATIONS/TEMP_FILES, needed for temporary files on SRM.
50	iSRMcatchall	unrecognized SRM error (shouldn't happen)
51	zmediumchanged	drive door opened; medium may have been changed.
52	endioerrs	placeholder for end of list.

File System Exports

The remainder of this chapter describes procedures and functions which are exported from the modules FS and MFS. These routines are normally called by compiler generated code to perform file operations (e.g. `read(textfile, size, color, description);`) and some string operations (e.g. `stread, strwrite`). Thus, these routines constitute the highest level of the workstation's file system support and are dependent on lower level support including the Directory Access Methods, the Access Methods, and the Transfer Methods (drivers).

For each routine described, the following information will be given.

The Pascal declaration of the routine.

Purpose: A brief description of what the routine is used for.

Parameters: A description of the parameters to the routine.

Stack: An illustration of the stack immediately after the routine is called (i.e. just after the `jsr` instruction). This may be useful if these routines are to be called by code written in assembly language. Parameters should be pushed as illustrated before the return address. The return address is normally pushed by a `jsr` instruction. Where a symbol similiar to this

```
----- <- sp + 10
| parameter  |//////////|
----- <- sp + 8
```

appears, it indicates that the parameter on the stack occupies only the most significant byte of the stack word.

Action: A brief description of the logic in the routine. In some cases this may be greatly abbreviated.

Errors: A summary of the expected error conditions encountered by the routine.

The following routines are explained.

doprefix	
fanonfile	
fblockio	[function]
fbufferref	[function]
fclose	
fcloseit	
feof	[function]
feoln	[function]
fget	
fgetxy	
fgotoxy	
fhopen	
fhpreset	
findvolume	[function]
finitb	
fixname	
fmaketype	
fmaxpos	[function]
foverfile	
foverprint	
fpage	
fposition	[function]
fput	
fread	
freadbool	
freadbytes	
freadchar	
freadenum	
freadint	
freadln	
freadpaoc	
freadreal	[MFS]
freadstr	
freadstrbool	
freadstrchar	
freadstrenum	
FREADSTRINT	[assembly]
freadstrpaoc	
freadstrreal	[MFS]
freadstrstr	
freadstrword	
freadword	

fseek
fstripname

fwrite
fwritebool
fwritebytes
fwritechar
writeenum
writeint
writeln
writepaoc
writereal [MFS]
writestr
writestrbool
writestrchar
writestrenum
FWRITESTRINT [assembly]
writestrpaoc
writestrreal [MFS]
writestrstr
writestrword
writeword

scantitle [function]

zapspace

```

procedure doprefix (var   dirname      : fid;
                   var   kvid         : vid;
                   var   kunit        : integer;
                   findunit          : boolean);

```

Purpose: To set the default (prefix) directory of a unit.
 Also returns the volume id and unit number of the unit.

Parameters: `dirname` volume id and path name.
`kvid` volume id returned.
`kunit` unit number returned.
`findunit` if true, directory must be present or return `ioresult` of `inounit`.

Stack:

```

----- <- sp + 18
| ptr to dirname          |
----- <- sp + 14
| ptr to kvid            |
----- <- sp + 10
| ptr to kunit          |
----- <- sp + 6
| findunit  |||////////|
----- <- sp + 4
| return address        |
----- <- stack pointer

```

Action: Call `scantitle` with `dirname`.
 Then call `findvolume`.
 If the unit is found and has a directory, call its DAM with a `setunitprefix` request and return the `uvid` and unit number.
 If the unit had no directory (specified as `#nn`) and `findunit` is false set `kvid` to `#nn` and return.
 Otherwise, set `ioresult` to indicate error condition.

Errors: `Scantitle` failed, set `ioresult` to `ibadtitle`.
`Findunit` true but no unit or directory found, set `ioresult` to `inounit`.
`Findunit` false and no directory or unit found but pathname followed colon, set `ioresult` to `ibadtitle`.
 The DAM request may fail and set `ioresult`.


```

procedure fanonfile (anyvar f          : fib;
                    var      name      : string;
                           kind       : filekind;
                           size       : integer);

```

Purpose: To open an anonymous file in a given directory (in name).

Parameters: f the FIB.
name this should include volume id but no filename is needed.
kind the file kind to be created.
size the size of the file to be created.

Stack:

```

----- <- sp + 20
| ptr to f |
----- <- sp + 16
| strmax(name) |//////////|
----- <- sp + 14
| ptr to name |
----- <- sp + 10
| kind |
----- <- sp + 8
| size |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Set fanonymous to true.
Call scantitle to extract fvid and ftitle from name.
Set fkind to kind.
Set fpos to size.
Set foptstring to addr(nullstring) (a dummy value).

If fpos > 0 then set fpos to fpos*fblksize.
Set feft to ehtable^[fkind];
Set fisnew to true.
Set freptcnt to 0.
Set fbufchanged to false.
Set flastpos to -1.
Set fstartaddress to 0.
Set pathid to -1.
Set fnosrmtemp to true.

Call findvolume with fvid and false (no verify) to get unit.
Call the DAM with a createfile request.
If ioresult is not inoerror, call findvolume with true (do verify) and call DAM with createfile request again.
Set fmodified to true.

Set up file state as follows.

fpos	0
fbufvalid	false
feof	true
freadmode	false
freadable	false
fwriteable	true
fleof	0
fmodified	true

Errors:

If name is too long or scantitle fails, set
ioresult to ibadtitle.

If findvolume returns unit 0, set ioreult to
inounit.

The DAM may set ioreult.

```

function fblockio (var f          : fib;
                  var buf        : window;
                  nblocks,
                  rblock        : integer;
                  doread        : boolean): integer;

```

Purpose: To read or write blocks of data on block boundaries. The read or write may be relative to current file position or from a specified position. Blocksize is fblksize = 512 bytes.

Parameters: f the FIB.
 buf the data buffer.
 nblocks the number of blocks to be read/written.
 rblock the starting file position (in blocks) (rblock < 0 indicates current file position).
 doread true if reading, false if writing.

Stack:

```

----- <- sp + 26
| function result          |
----- <- sp + 22
| ptr to f                 |
----- <- sp + 18
| ptr to buf               |
----- <- sp + 14
| nblocks                  |
----- <- sp + 10
| rblock                   |
----- <- sp + 6
| doread      |//////////|
----- <- sp + 4
| return address          |
----- <- stack pointer

```

Action: Calculate starting position as follows.
 If rblock >= 0 then set fpos to rblock*fblksize.
 If rblock < 0 then set fpos to fpos + (-fpos) mod fblksize (i.e. round fpos to block boundary).

Calculate number of bytes to be read/written as follows.
 blockbytes = nblocks*fblksize.
 If reading (doread = true), then if blockbytes > fleof - fpos (the number of bytes to the end of the file) then reduce blockbytes to fleof - fpos and reduce nblocks to (blockbytes + (fblksize-1)) div fblksize. (I.e. Don't attempt to read past end of file. Read to eof, and return nblocks as number of blocks that include all bytes to eof. The end of the last block will be uninitialized.)

Call the AM to read/write blockbytes bytes from/to the file at position fpos.

If the ioresult returned is inoerror, return nblocks.
Otherwise, return 0.

Errors: The AM may set ioresult.
If ioresult is not inoerror, 0 should be returned.

function fbufferref (var f : fib): windowp;

Purpose: To return a valid file window.

Parameters: f the FIB.

Stack:

```
----- <- sp + 12
| function result      |
----- <- sp + 8
| ptr to f            |
----- <- sp + 4
| return address      |
----- <- stack pointer
```

Action: If freadmode and not fbufvalid and flocked then call the AM to read frecsz bytes from the file at offset fpos into fwindow^. If the AM call resulted in an eof, set eof to true, and if feoln is not already true set fwindow^[0] to ' ', set feoln to true, set fbufvalid to true, and set ioresult to inoerror (i.e. create the 'GHOST' end of line). If the AM call did not result in an eof, set fbufvalid to true and feof to false. Return fwindow.

If the call to the AM was not necessary (i.e. the else for the first 'if' above), just return fwindow.

Errors: If not (freadable or fwriteable), set ioresult to inotopen. If not flocked, set ioresult to ifileunlocked. The AM call may set ioresult.

```

procedure fclose      (var      f          : fib;
                      ftype      : closetype);

```

Purpose: To close a file.

Parameters: f the FIB.
 ftype the type of close (i.e. cnormal, purge, lock, ccrunch).

Stack:

```

----- <- sp + 12
| ptr to f          |
----- <- sp + 8
| ftype            |
----- <- sp + 4
| return address   |
----- <- stack pointer

```

Action: If the file is not open (not freadable and not fwriteable) don't do anything.
 If fanonymous or ftype = purge or (fisnew and ftype = ccrunch) call the DAM with a purgefile request.

Otherwise, if the file is locked, call the AM to with a flush request, set the logical end of file to be the current file position if ftype = ccrunch (call DAM with stretchit request if this will extend the file) by setting fleof to fpos.
 Then call the DAM with a closefile request.

In any case, set the FIB fields to their default (closed) state:

```

freadmode      false
fbufvalid      false
freadable      false
fwriteable     false
flockable      false
flocked        true
feoln          true
feof           true

```

Errors: The DAM may set ioresult.
 The AM may set ioresult.
 In particular, the DAM may set ioresult to icantstretch.

```
procedure fcloseit (var f          : fib;
                   stype         : string255);
```

Purpose: To close a file.

Parameters: f the FIB.
stype the type of close. (This is a string as in the second parameter of a Pascal close call.)

Stack:

```
----- <- sp + 12
| ptr to f          |
----- <- sp + 8
| ptr to stype     |
----- <- sp + 4
| return address   |
----- <- stack pointer
```

Action: Convert stype to a closetype. Valid strings include 'NORMAL', 'TEMP', 'LOCK', 'SAVE', 'CRUNCH', and 'PURGE' ('NORMAL' is equivalent to 'TEMP' and 'LOCK' is equivalent to 'SAVE'). Case is ignored.

Call fclose with the closetype constructed.

Errors: If the file is not open (not freadable and not fwriteable), set ioreult to inotopen.
If the stype cannot be converted to a closetype, set ioreult to ibadclose.

function feof (var f : fib): boolean;

Purpose: To determine if file pointer is at end of file.

Parameters: f the FIB.

Stack:

```
----- <- sp + 10
| func. result |//////////|
----- <- sp + 8
| ptr to f      |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: If not (freadable or fwriteable) (i.e. file closed) then return true.
If frecsiz <= 0 (untyped files) then return true if fpos > fleof, false otherwise.
If frecsiz > 0 and freadable and fwriteable then return true if fposition(f) > fmaxpos(f), false otherwise.
Otherwise, call fbufferref if the unit is not interactive (to get proper FIB state), set ioresult to inoerror if it was ieof, and return f.feof.

Errors: If not flocked, set ioresult to ifileunlocked. Fbufferref may set an ioresult other than ieof.

function feoln (var f : fib): boolean;

Purpose: To determine if file pointer is at end of line.

Parameters: f the FIB.

Stack:

```
----- <- sp + 10
| func. result |//////////|
----- <- sp + 8
| ptr to f      |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Call fbufferref (to get proper FIB state), set ioreult to inoerror if it was ieof, and return f.feoln.

Errors: Fbufferref may set an ioreult other than ieof.

procedure fget (var f : fib);

Purpose: To position file pointer to next record.

Parameters: f the FIB.

Stack:

```
----- <- sp + 8
| ptr to f |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: If freadmode and not fbufvalid then call fread with f and f.fwindow^ to get the next record with the AM. Otherwise, set the lazy I/O condition by setting freadmode to true and fbufvalid to false.

Errors: If not (freadable or fwriteable), set ioresult to inotopen. If not freadable, set ioresult to inotreadable. If not flocked, set ioresult to ifileunlocked.

```

procedure fgetxy (anyvar f          : fib;
                 var      x,       : integer);
                 y

```

Purpose: To fetch the position of the cursor of an interactive file.

Parameters: f the FIB.
 x the x (column) coordinate of the cursor.
 y the y (row or line) coordinate of the cursor.

Stack:

```

----- <- sp + 16
| ptr to f          |
----- <- sp + 12
| ptr to x          |
----- <- sp + 8
| ptr to y          |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: Call the AM with a setcursor request.
 Set x to fxpos.
 Set y to fypos.

Errors: The AM may set ioreult (e.g. ibadrequest).

```

procedure fgotoxy (anyvar f          : fib;
                  x,                : integer);
                  y

```

Purpose: To position the cursor of an interactive file.

Parameters: f the FIB.
 x the x (column) coordinate of the cursor.
 y the y (row or line) coordinate of the cursor.

Stack:

```

----- <- sp + 16
| ptr to f          |
----- <- sp + 12
| x                 |
----- <- sp + 8
| y                 |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: Set fxpos to x.
 Set fypos to y.
 Call the AM with a setcursor request.

Errors: The AM may set ioreult (e.g. ibadrequest).

```

procedure fhopen (var f          : fib;
                  typ          : faccess;
                  var title,    : string255);
                  option

```

Purpose: To open a file with a name and an optional third parameter. (I.e. Pascal reset, open, append, rewrite procedures, e.g. reset(f,title,option).)

Parameters: f the FIB.
 typ the type of open (i.e. readonly = reset, readwrite = open, writeappend = append, writeonly = rewrite.)
 title the file identifier.
 option the equivalent of the third parameter in the Pascal open call.

Stack:

```

----- <- sp + 18
| ptr to f |
----- <- sp + 14
| typ |
----- <- sp + 12
| ptr to title |
----- <- sp + 8
| ptr to option |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Call fclose with f and cnormal.
 Set fanonymous to false.
 Call scantitle to extract fvid and ftitle and fkind and filesize from name.
 Set fpos to the filesize extracted.
 Set foptstring to addr(option).

 If fpos > 0 then set fpos to fpos*fblksize.
 Set feft to ehtable^[fkind];
 If typ = writeonly, set fisnew to true, false otherwise.

 Set freptcnt to 0.
 Set fbufchanged to false.
 Set flastpos to -1.
 Set fstartaddress to 0.
 Set pathid to -1.
 Set fnosrmtemp to true.

For typ = readonly (reset) do
 Call findvolume with fvid and false (no verify)
 to get unit.
 Call the DAM with a openfile request.
 If ioreult is not inoerror, call findvolume with
 true (do verify) and call DAM with openfile
 request again.

For typ = readwrite (open)
and
for typ = writeappend (append) do
 Call findvolume with fvid and false (no verify)
 to get unit.
 Call the DAM with an openfile request.
 If ioreult is not inoerror and not inofile, call
 findvolume with true (do verify) and call DAM
 with openfile request again.
 If it fails again, set fisnew to true (revert
 to fmaketype).

If openfile request succeeds, then if
fpos > fpeof, call the DAM with a stretchit
request.

If the stretchit request fails
(fpos still > fpeof), set ioreult to
icantstretch.

If fisnew (reverted to fmaketype), call the DAM
with a createfile request. If ioreult is not
inoerror, and the last findvolume call was not
with true (do verify), call findvolume with true
and call DAM with createfile request again.

Now that the hard stuff is done, if typ is
readwrite and fistextvar is true, call the DAM
to close the file and set ioreult to
ibadfiletype (not allowed to 'open' or 'append'
text files).

For typ = writeonly (rewrite) do
 Call findvolume with fvid and false (no verify)
 to get unit.
 Call the DAM with a createfile request.
 If ioreult is not inoerror, call findvolume with
 true (do verify) and call DAM with createfile
 request again.

Now for all values of typ set fmodified to fisnew.

Set up file state as follows.

```
For typ = readonly (reset)
    fpos          0
    fbufvalid     false
    feof          false
    freadmode     true
    feoln         true
    freadable     true
    fwriteable    false
```

```
For typ = readwrite (open)
    fpos          0
    fbufvalid     false
    feof          false
    freadmode     false
    freadable     true
    fwriteable    true
```

```
For typ = writeappend (append)
    fpos          fleof
    fbufvalid     false
    feof          true
    freadmode     false
    freadable     false
    fwriteable    true
```

```
For typ = writeonly (rewrite)
    fpos          0
    fbufvalid     false
    feof          true
    freadmode     false
    freadable     false
    fwriteable    true
    fleof         0
    fmodified     true
```

Errors: If name is too long or scantitle fails, set ioresult to ibadtitle.
If findvolume returns unit 0, set ioresult to inounit.
The DAM may set ioresult.

```

procedure fhpreset (var f : fib;
                   typ : faccess);

```

Purpose: To open (or reopen) a file without a name (i.e. no name was specified in the Pascal open call). If the file is already open, just change the state of the FIB.

Parameters: f the FIB.
 typ the type of open (i.e. readonly = reset, readwrite = open, writeappend = append, writeonly = rewrite.)

Stack:

```

----- <- sp + 10
| ptr to f |
----- <- sp + 6
| typ |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: If the file is not already open, it must be created as follows.
 Set fanonymous to true.
 Set fvid to the system volume.
 Set fkind to datafile.
 Set fpos to -1 (half the largest or second largest space, whichever is largest).
 Set foptstring to addr(nullstring) (dummy value).

 Set left to efttable^[fkind];
 Set fisnew to true.
 Set freptcnt to 0.
 Set fbufchanged to false.
 Set flastpos to -1.
 Set fstartaddress to 0.
 Set pathid to -1.
 Set fnosrmtemp to true.

 Call findvolume with fvid and false (no verify) to get unit.
 Call the DAM with a createfile request.
 If ioreult is not inoerror, call findvolume with true (do verify) and call DAM with createfile request again.
 Set fmodified to true.

Now for the old file or the one created above,
set up file state as follows.

```
For typ = readonly (reset)
      fpos          0
      fbufvalid    false
      feof         false
      freadmode    true
      feoln       true
      freadable    true
      fwriteable   false
```

```
For typ = readwrite (open)
      fpos          0
      fbufvalid    false
      feof         false
      freadmode    false
      freadable    true
      fwriteable   true
```

```
For typ = writeappend (append)
      fpos          fleof
      fbufvalid    false
      feof         true
      freadmode    false
      freadable    false
      fwriteable   true
```

```
For typ = writeonly (rewrite)
      fpos          0
      fbufvalid    false
      feof         true
      freadmode    false
      freadable    false
      fwriteable   true
      fleof        0
      fmodified    true
```

Errors: If name is too long or scantitle fails, set
 ioreresult to ibadtitle.
 If findvolume returns unit 0, set ioreresult to
 inounit.
 The DAM may set ioreresult.

```
function findvolume (var    fvid          : vid;
                    verify   : boolean): unitnum;
```

Purpose: To find the unit associated with the volume id fvid. With verify true or with fvid of the form '#nn' fvid is also set to the actual volume name.

Parameters: fvid the volume id.
verify boolean to indicate whether DAM must be called if searching by name.

Stack:

```
----- <- sp + 12
| function result      |
----- <- sp + 10
| ptr to fvid         |
----- <- sp + 6
| verify              |//////////|
----- <- sp + 4
| return address      |
----- <- stack pointer
```

Action: If fvid is of the form '#nn' then call the DAM for unit nn with a getvolumename request. If a name is returned by the DAM, return nn.

If fvid is not of the form '#nn' then

Search the unitable (unit 50 to unit 1 since faster devices (e.g. hard discs) are usually assigned to higher unit numbers) for uvid = fvid (uppercased if unit entry so indicates). If found and verify is true call the DAM for that unit with a getvolumename request. If the name returned by the DAM still matches fvid, set fvid to uvid and return the unit number.

If no match search again, but this time call DAM regardless of verify or matches to update uvid. If match is found set fvid to uvid and return the unit number as above.

Errors: Findvolume may return 0 if the volume is not found. Note that if fvid passed in is of the form '#nn' and unit nn has no volume name, findvolume will still return unit nn (not 0), but fvid will be unchanged.

```

procedure finitb (var f : fib;
                 window : windowp;
                 recbytes : integer);

```

Purpose: To initialize an FIB.

Parameters: f the FIB to be initialized.
 window the address for the file window.
 recbytes the file record size.
 Note: -3 is passed to indicate type text and to -1 to indicate an untyped file.

Stack:

```

----- <- sp + 16
| ptr to f |
----- <- sp + 12
| window |
----- <- sp + 8
| recbytes |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Set the following FIB fields to their default values:

```

freadmode      false
fbufvalid      false
freadable      false
fwriteable     false
flockable      false
flocked        true
feoln          true
feof           true

```

Set fwindow to window.
 Set fistextvar to true if recbytes = -3,
 false otherwise.

If recsize = -1 (untyped file) then set fwindow to nil and set frecsize to 0.

If recsize <= 0 (except -1) set frecsize to 1 and initialize the first character of fwindow[^] to chr(0).

Otherwise, set frecsize to recbytes.

Set fbuffered to true if frecbytes > 0,
 false otherwise.

Errors: None.

```
procedure fixname (var title : string;
                  kind : filekind);
```

Purpose: To put proper suffixes on file names. Also removes spaces and control characters.

Parameters: title the file name to be fixed.
kind the file type associated with the suffix.

Stack:

```
----- <- sp + 12
| strmax(title)|//////////|
----- <- sp + 10
| ptr to title |
----- <- sp + 6
| kind         |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Call zapspaces with title.
If title ends in ':' then do nothing.
If title ends in '.' then remove last character.
Otherwise, if a call to suffix with title returns datafile (i.e. no suffix is already present) then look up a suffix in suffixtable (indexed by kind), and (if it will fit) append the suffix to title. If the suffix does not fit do nothing.

Errors: None.

```

procedure fmakefile (anyvar f          : fib;
                    var title,
                        option,
                        typestring     : string);

```

Purpose: To make a file of a given type (i.e. disregard suffix of title).

Parameters: f the FIB.
title the file identifier.
option the equivalent of the third parameter in the Pascal open call.
typestring a string with a suffix corresponding to the file type desired.

Stack:

```

----- <- sp + 26
| ptr to f |
----- <- sp + 22
| strmax(title)|//////////|
----- <- sp + 20
| ptr to title |
----- <- sp + 16
|strmax(option)|//////////|
----- <- sp + 14
| ptr to option |
----- <- sp + 10
|strmax(typestring)|//////////|
----- <- sp + 8
| ptr to typestring |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Call fclose with f and cnormal.
Set fanonymous to false.
Call scantitle to extract fvid and ftitle and filesize from name.
Set fpos to the filesize extracted.
Call suffix with typestring and set fkind to the kind returned.
Set foptstring to addr(option).

If fpos > 0 then set fpos to fpos*fblksize.
Set left to ehtable^[fkind];
Set fisnew to true.
Set freptcnt to 0.
Set fbufchanged to false.
Set flastpos to -1.
Set fstartaddress to 0.
Set pathid to -1.
Set fnosrmtemp to true.

Call findvolume with fvid and false (no verify) to get unit.
Call the DAM with a createfile request.
If ioreult is not inoerror, call findvolume with true (do verify) and call DAM with createfile request again.
Set fmodified to true.

Set up file state as follows.

fpos	0
fbufvalid	false
feof	true
freadmode	true
freadable	false
fwriteable	true
fleof	0
fmodified	true

Errors: If name is too long or scantitle fails, set ioreult to ibadtitle.
If findvolume returns unit 0, set ioreult to inounit.
The DAM may set ioreult.

function fmaxpos (var f : fib): integer;

Purpose: To determine the number of records in a file.

Parameters: f the FIB.

Stack:

```
----- <- sp + 12
| function result      |
----- <- sp + 8
| ptr to f            |
----- <- sp + 4
| return address      |
----- <- stack pointer
```

Action: Return fleof div frecsiz.

Errors: Return 0 on all errors.
If not (freadable or fwriteable), set ioresult to inotopen.
If not (freadable and fwriteable), set ioresult to inotdirect.
If not flocked, set ioresult to ifileunlocked.

```

procedure foverfile (anyvar f          : fib;
                    var title,
                        option,
                        typestring    : string);

```

Purpose: To create a file of a given type (i.e. disregard suffix of title) which will 'overwrite' another file of the same name.

Parameters: f the FIB.
title the file identifier.
option the equivalent of the third parameter in the Pascal open call.
typestring a string with a suffix corresponding to the file type desired.

Stack:

```

----- <- sp + 26
| ptr to f |
----- <- sp + 22
| strmax(title) |
----- <- sp + 20
| ptr to title |
----- <- sp + 16
|strmax(option) |
----- <- sp + 14
| ptr to option |
----- <- sp + 10
|strmax(typestring) |
----- <- sp + 8
| ptr to typestring |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Call fclose with f and cnormal.
Set fanonymous to false.
Call scantitle to extract fvid and ftitle and filesize from name.
Set fpos to the filesize extracted.
Call suffix with typestring and set fkind to the kind returned.
Set foptstring to addr(option).

If fpos > 0 then set fpos to fpos*fblksize.
Set feft to efttable^[fkind];
Set fisnew to false.
Set freptcnt to 0.
Set fbufchanged to false.
Set flastpos to -1.
Set fstartaddress to 0.
Set pathid to -1.
Set fnosrmtemp to true.

Call findvolume with fvid and false (no verify) to get unit.

Call the DAM with an overwritefile request.

If ioresult is not inoerror and not inofile, call findvolume with true (do verify) and call DAM with overwritefile request again.

If it fails again, set fisnew to true (revert to fmake type).

If overwritefile request succeeds, then if fpos > fpeof, call the DAM with a stretchit request.

If the stretchit request fails (fpos still > fpeof), call the DAM with a purgefile request to clean up the temporary file and set ioresult to icantstretch.

If fisnew (reverted to fmake type), call the DAM with a createfile request. If ioresult is not inoerror, and the last findvolume call was not with true (do verify), call findvolume with true and call DAM with createfile request again.

Set fmodified to true.

Set up file state as follows.

fpos	0
fbufvalid	false
feof	true
freadmode	true
freadable	false
fwriteable	true
fleof	0
fmodified	true

Errors: If name is too long or scantitle fails, set ioresult to ibadtitle.
If findvolume returns unit 0, set ioresult to inounit.
The DAM may set ioresult.

```
procedure foverprint(var t : text);
```

Purpose: To write an overprint command to a text file.

Parameters: t the text file.

Stack:

```
----- <- sp + 8  
| ptr to t |  
----- <- sp + 4  
| return address |  
----- <- stack pointer
```

Action: Write an eol (cr -- chr(13)) to the text file.
For printer files this will reposition the print
head to the beginning of the current print line.

Errors: None.

```
procedure fpage (var t : text);
```

Purpose: To write a page eject sequence to a text file.

Parameters: t the text file.

Stack:

```
----- <- sp + 8  
| ptr to t |  
----- <- sp + 4  
| return address |  
----- <- stack pointer
```

Action: Write an eol (cr -- chr(13)) and clearscr
(ff -- chr(12)) to the text file.

Errors: None.

function fposition (var f : fib): integer;

Purpose: To determine the position of the file pointer.

Parameters: f the FIB.

Stack:

```
----- <- sp + 12
| function result      |
----- <- sp + 8
| ptr to fib          |
----- <- sp + 4
| return address      |
----- <- stack pointer
```

Action: Return fpos div frecsz + 1 - ord(fbval).

Errors: Return 0 on all errors.
If not (freadable or fwriteable), set ioresult to inotopen.
If not flocked, set ioresult to ifileunlocked.

procedure fput (var f : fib);

Purpose: To write the file window to the file.

Parameters: f the FIB.

Stack:

```
----- <- sp + 8
| ptr to f |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Call fwrite with f and f.fwindow^.

Errors: See fwrite.

```

procedure fread      (anyvar f          : fib;
                     anyvar buf       : window);

```

Purpose: To read a record from the file.

Parameters: f the FIB.
 buf the buffer to read the record into.

Stack:

```

----- <- sp + 12
| ptr to f          |
----- <- sp + 8
| ptr to buf       |
----- <- sp + 4
| return address   |
----- <- stack pointer

```

Action: If fbufvalid and flocked then move frecsz bytes from fwindow[^] to buf and set lazy I/O condition by setting fbufvalid to false.

Otherwise, call the AM with a readbytes request to read frecsz bytes into buf from the file at position fpos.
 Set lazy I/O condition by setting freadmode to true and fbufvalid to false.

If the AM call resulted in an eof and fistextvar and not feoln then set buf[0] to ' ', set feoln to true, and set ioresult to inoerror (i.e. create the 'GHOST' end of line).

Errors: If not (freadable or fwriteable), set ioresult to inotopen.
 If not freadable, set ioresult to inotreadable.
 If not flocked, set ioresult to ifileunlocked.

```
procedure freadbool (var t : text;
                    var b : boolean);
```

Purpose: To read (formatted) a boolean from a text file.
(I.e. read an identifier and return the boolean value.)

Parameters: t the text file.
b the boolean value to be returned.

Stack:

```
----- <- sp + 12
| ptr to t |
----- <- sp + 8
| ptr to b |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Call freadenum with the address of a constant table of string values for the enumerated type (FALSE, TRUE).
If the index (scalar) returned is 1, set b to true.
Otherwise set b to false.

Errors: Freadenum may set ioresult.

```

procedure freadbytes(anyvar f           : fib;
                    anyvar buf         : window;
                    size              : integer);

```

Purpose: To read size bytes from buf to the file.

Parameters: f the FIB.
 buf the buffer to be read into.
 size the number of bytes to be read.

Stack:

```

----- <- sp + 16
| ptr to f           |
----- <- sp + 12
| ptr to buf        |
----- <- sp + 8
| size              |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: Call the AM with a readbytes request to read size bytes into buf from the file at position fpos. Set lazy I/O condition by setting freadmode to true and fbufvalid to false.

Errors: If not (freadable or fwriteable), set ioresult to inotopen.
 If not freadable, set ioresult to inotreadable.
 If not flocked, set ioresult to ifileunlocked.
 The AM may set ioresult.


```
procedure freadchar (var t : text;
                    var ch : char);
```

Purpose: To read a character from a text file.

Parameters: t the text file.
ch the character to be read into.

Stack:

-----	<- sp + 12
ptr to t	
-----	<- sp + 8
ptr to ch	
-----	<- sp + 4
return address	
-----	<- stack pointer

Action: Call fread with t and ch.

Errors: See fread.

```

procedure freadenum (var t      : text;
                    var i      : shortint;
                    p          : vptr);

```

Purpose: To read (formatted) an enumerated type from a text file.
(I.e. read an identifier and return the scalar value.)

Parameters: t the text file.
i the index into p.
p the compiler generated table of string values for the enumerated type.

Stack:

```

----- <- sp + 16
| ptr to t |
----- <- sp + 12
| ptr to i |
----- <- sp + 8
| p |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Using t^ (compiler generated call to fbufferref), get (compiler generated call to fget), read (handle backspace and clearline if unit is interactive) and ignore all leading spaces, and read all legal identifier characters ('0' to '9', 'A' to 'Z', 'a' to 'z' and '_' starting with 'A' to 'Z', 'a' to 'z') up to the first illegal character into a string (max of 255 characters). Call freadstrenum to search the compiler generated table for the constructed string and return the index in i.

Errors: T^ (compiler generated call to fbufferref) may set iresult.
Get (compiler generated call to fget) may set iresult.
Freadstrenum may set iresult or escape on failures.

```
procedure freadint (var t : text;
                   var i : integer);
```

Purpose: To read (formatted) an integer from a text file.

Parameters: t the text file.
i the integer to be read into.

Stack:

```
----- <- sp + 12
| ptr to t |
----- <- sp + 8
| ptr to i |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Using t[^] (compiler generated call to fbufferref),
get (compiler generated call to fget), read
(handle backspace and clearline if the unit is
interactive) and ignore all leading spaces, and
read at most 1 sign (+,-) character and all digit
('0' to '9') characters up to the next non digit
into a string (max of 255 characters).
Call stread (compiler generated call to
freadstrint) to convert the constructed string to
an integer (i).

Errors: T[^] (compiler generated call to fbufferref) may
set ioreult.
Get (compiler generated call to fget) may
set ioreult.
Freadstrint may set ioreult or escape on failures.

```
procedure freadln (var t : text);
```

Purpose: To read (and ignore) characters from a text file up to and including the next end of line marker.

Parameters: t the text file.

Stack:

```
----- <- sp + 8  
| ptr to t |  
----- <- sp + 4  
| return address |  
----- <- stack pointer
```

Action: While the text file is not at end of line, do gets on the text file to skip characters (handle backspace and clearline if unit is interactive). Do one more get on the text file to consume the end of line marker.

Errors: None.

```

procedure freadpaoc (var t      : text;
                    var a      : window;
                    aleng      : shortint);

```

Purpose: To read (formatted) a packed array of characters from a text file.

Parameters: t the text file.
a the array to be read into.
aleng the size of the array.

Stack:

```

----- <- sp + 14
| ptr to t |
----- <- sp + 10
| ptr to a |
----- <- sp + 6
| aleng |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: If the unit is interactive then using t^ (compiler generated call to fbufferref), get (compiler generated call to fget), read (handle backspace and clearline if unit is interactive) a maximum of aleng characters into a starting at a[1] until the file is at end of line.

If the unit is not interactive (no need to read one character at a time to handle backspace and clearline), read and save the look ahead character, call the AM with a readtoeol request to read a maximum of a aleng-1 byte string into s starting at s[1] (this means a[1] will hold the string length) and place the saved look ahead character (see above) in a[1]. This may seem round about but much of the code is also used in reading strings.

Fill the rest of the array with spaces.

Errors: T^ (compiler generated call to fbufferref) may set ioreult.
Get (compiler generated call to fget) may set ioreult.
AM may set ioreult.

```
procedure freadreal    (var    t    : text;
                       var    x    : real);
```

Purpose: To read (formatted) a real from a text file.

Parameters: t the text file.
x the real to be read into.

Stack:

```
----- <- sp + 12
| ptr to t |
----- <- sp + 8
| ptr to x |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Using t[^] (compiler generated call to fbufferref), get (compiler generated call to fget), read (handle backspace and clearline if the unit is interactive) and ignore all leading spaces, and read all valid characters that make up a real number representation (e.g. sign, digits, decimal point, exponent field) up to the next invalid character into a string (max of 255 characters). Call strread (compiler generated call to freadstreal) to convert the constructed string to a real (x).

Errors: T[^] (compiler generated call to fbufferref) may set iresult.
Get (compiler generated call to fget) may set iresult.
If t is at end of file, set iresult to ieof.
Freadstreal may set iresult on failures.

```

procedure freadstr (var t          : text;
                   var s          : string);

```

Purpose: To read (formatted) a string from a text file.

Parameters: t the text file.
s the string to be read into.

Stack:

```

----- <- sp + 14
| ptr to t          |
----- <- sp + 10
| strmax(s)  |//////////|
----- <- sp + 8
| ptr to s          |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: If the unit is interactive then using t^ (compiler generated call to fbufferref), get (compiler generated call to fget), read (handle backspace and clearline if unit is interactive) a maximum of 255 characters into s starting at s[1] until the file is at end of line.
Set the length of the string to the number of characters read.

If the unit is not interactive (no need to read one character at a time to handle backspace and clearline), read and save the look ahead character, call the AM with a readtoeol request to read a maximum of a 254 byte string into s starting at s[1] (this means s[1] will hold the string length), set the length of s to s[1]+1, and place the saved look ahead character (see above) in s[1]. This may seem round about but much of the code is also used in reading packed arrays of characters.

Errors: T^ (compiler generated call to fbufferref) may set ioresult.
Get (compiler generated call to fget) may set ioresult.
AM may set ioresult.

```

procedure freadstrbool(var s           : string255;
                      var p2         : integer;
                      var b           : boolean);

```

Purpose: To read (formatted) a boolean from a string.
(I.e. read an identifier and return the boolean value.)

Parameters: s the string.
p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
b the boolean value to be returned.

Stack:

```

----- <- sp + 16
| ptr to s           |
----- <- sp + 12
| ptr to p2          |
----- <- sp + 8
| ptr to b           |
----- <- sp + 4
| return address     |
----- <- stack pointer

```

Action: Call freadstrenum with the address of a constant table of string values for the enumerated type (FALSE,TRUE).
If the index (scalar) returned is 1, set b to true.
Otherwise set b to false.

Errors: Freadstrenum may set ioreult.


```

procedure freadstrchar(var s      : string255;
                      var p2     : integer;
                      var ch     : char);

```

Purpose: To read a character from a string.

Parameters: s the string.
 p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
 ch the character to be read into.

Stack:

```

----- <- sp + 16
| ptr to s          |
----- <- sp + 12
| ptr to p2        |
----- <- sp + 8
| ptr to ch        |
----- <- sp + 4
| return address   |
----- <- stack pointer

```

Action: Set ch to s[p2].
 Increment p2 by 1.

Errors: If p2 < 1 or p2 > the length of s, set ioreult to istrovfl.

```

procedure freadstrenum(var s      : string255;
                      var p2    : integer;
                      var i      : shortint;
                      p          : vptr);

```

Purpose: To read (formatted) an enumerated type from a string.
(I.e. read an identifier and return the scalar value.)

Parameters: s the string.
p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
i the index into p.
p the compiler generated table of string values for the enumerated type.

Stack:

```

----- <- sp + 20
| ptr to s |
----- <- sp + 16
| ptr to p2 |
----- <- sp + 12
| ptr to i |
----- <- sp + 8
| p |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Starting at s[p2] skip all leading spaces and copy all legal identifier characters ('0' to '9', 'A' to 'Z', 'a' to 'z' and '_' starting with 'A' to 'Z', 'a' to 'z') up to the first illegal character into a string (max of 80 characters). Add the number of characters skipped and copied to p2.
Search the table p for the identifier and set i to the index.

Errors: If p2 < 1 or p2 > length of s, set iresult to istrovfl.
If the leading spaces extend to the end of s, set iresult to istrovfl.
If the identifier does not start with 'A' to 'Z', 'a' to 'z', set iresult to ibadformat.
If the identifier is not found in the table p, set iresult to ibadformat.

```
{PROCEDURE FREADSTRINT(VAR S : STRING255;
VAR P2, I : INTEGER);}
```

Purpose: To read (formatted) an integer from a string.

Parameters: s the string.
p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
i the integer to be read into.

Stack:

```
----- <- sp + 16
| ptr to s |
----- <- sp + 12
| ptr to p2 |
----- <- sp + 8
| ptr to i |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: This routine is written in assembly for speed.

Starting at s[p2], skip spaces.
If first non space is a sign remember it.
Initialize an accumulator value to 0.
While the next character is a digit get characters one at a time from s, and, for each one, multiply the accumulator by ten and add the character's numerical value to it.
Add the number of characters skipped and used to p2.
Adjust the sign of the accumulator and assign it to i.

Errors: If p2 < 1 or p2 > length of s, set ioresult to ibadformat.
If not at least on digit, set ioresult to ibadformat.
If number is too large (overflow), set ioresult to ibadformat.

```

procedure freadstrpaoc(var s           : string255;
                       var p2        : integer;
                       var a         : window;
                       aleng         : shortint);

```

Purpose: To read (formatted) a packed array of characters from a string.

Parameters: s the string.
p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
a the packed array of characters to be written.
aleng the size of the packed array of characters.

Stack:

```

----- <- sp + 18
| ptr to s           |
----- <- sp + 14
| ptr to p2         |
----- <- sp + 10
| ptr to a          |
----- <- sp + 6
| aleng             |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: Initialize a to all spaces.
Copy characters from s starting at s[p2] to a until aleng characters have been copied or until there are no more characters in s.
Add the number of characters copied to p2.

Errors: If p2 < 1 or p2 > length of s, set ioresult to istrovfl.

```

procedure freadstreal (var    s      : string255;
                      var    p2     : integer;
                      var    x      : real);

```

Purpose: To read (formatted) a real from a string.

Parameters: s the string.
 p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
 x the real to be read into.

Stack:

```

----- <- sp + 16
| ptr to s          |
----- <- sp + 12
| ptr to p2        |
----- <- sp + 8
| ptr to x         |
----- <- sp + 4
| return address   |
----- <- stack pointer

```

Action: Starting at s[p2], skip spaces.
 Read all valid characters for a real number (e.g. sign, digits, decimal point, exponent field) up to the next invalid character.
 Convert the characters read into a real (x).
 Set p2 to one past the last character read.

Errors: If p2 < 1 or p2 > length of s, set ioresult to strovfl.
 If there is no valid real represented by the characters, set ioresult to ibadformat.

```

procedure freadstrstr(var t : string255;
                     var p2 : integer;
                     var s : string);

```

Purpose: To read (formatted) a string from a string.

Parameters: t the string to be read from.
 p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
 s the string to be read into.

Stack:

```

----- <- sp + 18
| ptr to t |
----- <- sp + 14
| ptr to p2 |
----- <- sp + 10
| strmax(s) |//////////|
----- <- sp + 8
| ptr to s |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Copy characters from t starting at t[p2] into s starting at s[1] until strmax(s) characters are copied or until there are no more characters in t to copy.
 Add the number of characters copied to p2.

Errors: If p2 < 1 or p2 > length of t, set ioresult to istrovfl.

```

procedure freadstrword(var s      : string255;
                      var p2    : integer;
                      var i      : shortint);

```

Purpose: To read (formatted) a short (16 bit) integer from a string.

Parameters: s the string.
 p2 the index into the string. Initially where the read is to start. Finally one past the last character read.
 i the short integer to be read into.

Stack:

```

----- <- sp + 16
| ptr to s |
----- <- sp + 12
| ptr to p2 |
----- <- sp + 8
| ptr to i |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Call stread (compiler generated call to freadstrint) to read an integer from the string. If the integer is not in the range -32768 to 32767 then escape(-8). Otherwise, set i to the integer read.

Errors: Freadstrint may set ioreult or escape on failures. If the integer is not in range, escape(-8).

```
procedure freadword (var t : text;
                    var i : shortint);
```

Purpose: To read (formatted) a short (16 bit) integer from a text file.

Parameters: t the text file.
i the short integer to be read into.

Stack:

```
----- <- sp + 12
| ptr to t |
----- <- sp + 8
| ptr to i |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Read (compiler generated call to freadint) and integer (32 bit) from t.
If the integer is not in the range -32768 to 32767 then escape(-8).
Otherwise, set i to the integer read.

Errors: The integer is out of range, escape(-8).
Freadint may set ioreult.


```
procedure fseek      (var      f          : fib;
                     position    : integer);
```

Purpose: To reposition the file pointer.

Parameters: f the FIB.
position the desired record position of
the pointer.

Stack:

```
----- <- sp + 12
| ptr to f          |
----- <- sp + 8
| position          |
----- <- sp + 4
| return address    |
----- <- stack pointer
```

Action: If position < 1 then set fpos to 0.
Otherwise, set fpos to (position - 1) * frecsiz.
Put file in non read mode condition by setting
freadmode to false and fbufvalid to false.

Errors: If not (freadable or fwriteable), set ioresult
to inotopen.
If not (freadable and fwriteable), set ioresult
to inotdirect.
If not flocked, set ioresult to ifileunlocked.

```

procedure fstripname(      s          : fid;
                          var    pvname,
                                ppath,
                                pfname : string);

```

Purpose: To remove passwords from file identifiers.

Parameters: s the file identifier.
pvname the volume name returned.
ppath the path name returned.
pfname the file name returned.

Stack:

```

----- <- sp + 26
| ptr to s |
----- <- sp + 22
|strmax(pvname)|//////////|
----- <- sp + 20
| ptr to pvname |
----- <- sp + 16
| strmax(ppath)|//////////|
----- <- sp + 14
| ptr to ppath |
----- <- sp + 10
|strmax(pfname)|//////////|
----- <- sp + 8
| ptr to pfname |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Scantitle is called with the fid passed in.
Then findvolume is called to find the volume indicated by the fid.
Then the DAM for that unit is called with the fid.
The DAM then parses the fid into three parts: volume name, pathname, and file name without passwords.

Errors: If scantitle fails ioresult is set to ibadtitle.
If findvolume fails, ioresult is set to inunit.
Otherwise, the DAM may set the ioresult as appropriate (e.g. ibadtitle, etc...).

```

procedure fwrite (anyvar f          : fib;
                 anyvar buf        : window);

```

Purpose: To write a record to the file.

Parameters: f the FIB.
 buf the record to be written.

Stack:

```

----- <- sp + 12
| ptr to f          |
----- <- sp + 8
| ptr to buf       |
----- <- sp + 4
| return address   |
----- <- stack pointer

```

Action: Call the AM with a writebytes request to write
 frcsize bytes from buf at position fpos.
 Set non read mode condition by setting
 freadmode to false and fbufvalid to false.

Errors: If not (freadable or fwriteable), set ioreult
 to inotopen.
 If not fwriteable, set ioreult to inotwriteable.
 If not flocked, set ioreult to ifileunlocked.
 The AM may set ioreult.

```

procedure fwritebool(var      t          : text;
                    b        : boolean;
                    rleng    : shortint);

```

Purpose: To write (formatted) a boolean to a text file.
(I.e. write an identifier given the boolean value.)

Parameters: t the text file.
b the boolean value.
rleng the field width to be written into
(max 255).

Stack:

```

----- <- sp + 12
| ptr to t                               |
----- <- sp + 8
| b          |////////////////////|
----- <- sp + 6
| rleng      |
----- <- sp + 4
| return address
----- <- stack pointer

```

Action: Call fwriteenum with the ordinal value of b as the scalar, the address of a constant table of string values for the enumerated type (FALSE,TRUE) and rleng as the field width.

Errors: Fwriteenum may set ioreult.

```

procedure fwritebytes(anyvar f          : fib;
                     anyvar buf       : window;
                     size              : integer);

```

Purpose: To write size bytes from buf to the file.

Parameters: f the FIB.
 buf the buffer to be written.
 size the number of bytes to be written.

Stack:

```

----- <- sp + 16
| ptr to f          |
----- <- sp + 12
| ptr to buf       |
----- <- sp + 8
| size             |
----- <- sp + 4
| return address   |
----- <- stack pointer

```

Action: Call the AM with a writebytes request to write size bytes from buf at position fpos. Set non read mode condition by setting freadmode to false and fbufvalid to false.

Errors: If not (freadable or fwriteable), set ioreult to inotopen.
 If not fwriteable, set ioreult to inotwriteable.
 If not flocked, set ioreult to ifileunlocked.
 The AM may set ioreult.

```

procedure fwritechar(var      t           : text;
                    ch       : char;
                    rlen     : shortint);

```

Purpose: To write (formatted) a character to a text file.

Parameters: t the text file.
 ch the character to be written.
 rlen the field width to be written into
 (max 255).

Stack:

```

----- <- sp + 12
| ptr to t                               |
----- <- sp + 8
| ch          |////////////////////|
----- <- sp + 6
| rlen                               |
----- <- sp + 4
| return address                       |
----- <- stack pointer

```

Action: If rlen < 1 then set rlen to 1.
 Construct a packed array of characters of rlen-1
 spaces followed by the character ch.
 Call fwritebytes to write rlen characters from
 the packed array to the text file.

Errors: Rlen > 255 will cause boundary error.
 Fwritebytes may set ioreult.

```

procedure fwriteenum(var      t          : text;
                    i          : shortint;
                    rleng     : shortint;
                    p          : vptr);

```

Purpose: To write (formatted) an enumerated type to a text file.
(I.e. write an identifier given the scalar value.)

Parameters: t the text file.
i the index into p (the scalar value).
rleng the field width to be written into (max 255).
p the compiler generated table of string values for the enumerated type.

Stack:

```

----- <- sp + 16
| ptr to t          |
----- <- sp + 12
| i                 |
----- <- sp + 10
| rleng             |
----- <- sp + 8
| p                 |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: Call fwritestrenum to write the identifier to a string (i.e. to do the hard part).
If ioreult is inoerror then call fwritebytes to write the string to the file.

Errors: Fwritestrenum may set ioreult.
Fwritebytes may set ioreult.

```
procedure fwriteint (var      t          : text;
                    i          : integer;
                    rleng     : shortint);
```

Purpose: To write (formatted) a integer to a text file.

Parameters: t the text file.
i the integer to be written.
rleng the field width to be written into
(max 255).

Stack:

```
----- <- sp + 14
| ptr to t          |
----- <- sp + 10
| i                 |
----- <- sp + 6
| rleng             |
----- <- sp + 4
| return address    |
----- <- stack pointer
```

Action: Call strwrite (compiler generated call to
fwritestrnt) to write the integer to a string.
If ioresult is inoerror, call fwritebytes to write
the string to the text file.

Errors: Fwritestrnt may set ioresult.
Fwritebytes may set ioresult.


```
procedure fwriteln (var f : fib);
```

Purpose: To write an end of line marker to the file.

Parameters: f the FIB.

Stack:

```
----- <- sp + 8  
| ptr to f |  
----- <- sp + 4  
| return address |  
----- <- stack pointer
```

Action: Call the AM with a writeeol request at position fpos.
Set non read mode condition by setting freadmode to false and fbufvalid to false.

Errors: If not (freadable or fwriteable), set ioreult to inotopen.
If not fwriteable, set ioreult to inotwriteable.
If not flocked, set ioreult to ifileunlocked.
The AM may set ioreult.

```

procedure fwritepaoc(var t          : text;
                    var a          : window;
                    aleng,
                    rleng          : shortint);

```

Purpose: To write (formatted) a packed array of characters to a text file.

Parameters: t the text file.
a the string to be written (max length of 80).
aleng the size of the array.
rleng the field width to be written into.
Note that this must be no more than 255 + aleng.

Stack:

```

----- <- sp + 16
| ptr to t          |
----- <- sp + 12
| ptr to a          |
----- <- sp + 8
| aleng             |
----- <- sp + 6
| rleng             |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: If rleng < 0 then set rleng to the length of s.
If rleng > aleng, call fwritechar to write a space in a field width of rleng - aleng (i.e. write that many spaces) and set rleng to the aleng.

Call fwritebytes to write aleng bytes from the packed array of characters to the file.

Errors: Fwritechar may set ioreult.
Fwritebytes may set ioreult.

```

procedure fwritereal (var t : text;
                    x : real;
                    w,
                    d : shortint);

```

Purpose: To write (formatted) a real to a text file.

Parameters: t the text file.
x the real to be written.
w the field width to be written into (max 255).
d the number of digits after the decimal point.

Stack:

```

----- <- sp + 16
| ptr to t |
----- <- sp + 12
| ptr to x |
----- <- sp + 8
| w |
----- <- sp + 6
| d |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Call strwrite (compiler generated call to fwritestrreal) to write the real to a string. If ioreult is inoerror, call fwritebytes to write the string to the text file.

Errors: Fwritestrreal may set ioreult.
Fwritebytes may set ioreult.

```

procedure fwritestr (var t          : text;
                    anyvar s       : string80;
                    rleng          : shortint);

```

Purpose: To write (formatted) a string to a text file.

Parameters: t the text file.
s the string to be written (max length of 80).
rleng the field width to be written into.
Note that this must be no more than
255 + length of s.

Stack:

```

----- <- sp + 14
| ptr to t          |
----- <- sp + 10
| ptr to s          |
----- <- sp + 6
| rleng             |
----- <- sp + 4
| return address    |
----- <- stack pointer

```

Action: If `rleng < 0` then set `rleng` to the length of `s`.
If `rleng > length of s`, call `fwritechar` to write
a space in a field width of `rleng - length of s`
(i.e. write that many spaces) and set `rleng` to
the length of `s`.

Call `fwritebytes` to write the string to the file.

Errors: `Fwritechar` may set `ioresult`.
`Fwritebytes` may set `ioresult`.

```

procedure fwritestrbool(var s           : string;
                        var p2         : integer;
                        b              : boolean;
                        rleng          : shortint);

```

Purpose: To write (formatted) a boolean to a string.
(I.e. write an identifier given the boolean value.)

Parameters: s the string.
p2 the index into the string. Initially where the write is to start. Finally one past the last character written.
b the boolean value.
rleng the field width to be written into (max 255).

Stack:

```

----- <- sp + 18
| strmax(s)  |//////////|
----- <- sp + 16
| ptr to s  |          |
----- <- sp + 12
| ptr to p2 |          |
----- <- sp + 8
| b         |//////////|
----- <- sp + 6
| rleng    |          |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Call fwritestrenum with the ordinal value of b as the scalar, the address of a constant table of string values for the enumerated type (FALSE,TRUE) and rleng as the field width.

Errors: Fwritestrenum may set ioresult.

```

procedure fwritestrchar(var s           : string;
                        var p2         : integer;
                        ch             : char;
                        rleng          : shortint);

```

Purpose: To write (formatted) a character into a string.

Parameters: s the string.
p2 the index into the string. Initially where the write is to start. Finally one past the last character written.
ch the character to be written.
rleng the field width to be written into.

Stack:

```

----- <- sp + 18
| strmax(s)  |//////////|
----- <- sp + 16
| ptr to s   |
----- <- sp + 12
| ptr to p2  |
----- <- sp + 8
| ch        |//////////|
----- <- sp + 6
| rleng     |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Convert ch to a string of length 1.
If rleng < 1 then set rleng to 1.
Call fwritestrstr with s, p2, the constructed string, and rleng.

Errors: Fwritestrstr may set ioreult.

```

procedure fwritestrenum(var s           : string;
                       var p2         : integer;
                       i,
                       rleng          : shortint;
                       p               : vptr);

```

Purpose: To write (formatted) an enumerated type to a string.
(I.e. write an identifier given the scalar value.)

Parameters:

- s the string.
- p2 the index into the string. Initially where
 the write is to start. Finally one past
 the last character written.
- i the index into p (the scalar value).
- rleng the field width to be written into
 (max 255).
- p the compiler generated table of string
 values for the enumerated type.

Stack:

```

----- <- sp + 22
| strmax(s)  |//////////|
----- <- sp + 20
| ptr to s   |
----- <- sp + 16
| ptr to p2  |
----- <- sp + 12
| i          |
----- <- sp + 10
| rleng      |
----- <- sp + 8
| p          |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Extract the identifier in table p indexed by i.
Call fwritestrstr with s, p2, the identifier, and
rleng.

Errors: If the index i is out of the range of the table,
escape(-8).
Fwritestrstr may set ioresult.

```
{PROCEDURE FWRITESTRINT(VAR T           : STRING;
                        VAR P2          : INTEGER;
                        I               : INTEGER;
                        RLENG           : SHORTINT); }
```

Purpose: To write (formatted) an integer to a string.

Parameters: t the string.
 p2 the index into the string. Initially where the write is to start. Finally one past the last character written.
 i the integer to be written.
 rlung the field width to be written into.

Stack:

```
----- <- sp + 20
| strmax(t) |//////////|
----- <- sp + 18
| ptr to t   |
----- <- sp + 14
| ptr to p2  |
----- <- sp + 10
| i          |
----- <- sp + 6
| rlung      |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: This routine is written in assembly for speed.

Remember sign of i.
 By successively dividing by decreasing powers of ten the remainder of previous divisions, determine the digits in order (left to right) and put them into a dummy string.

If rlung > length of the dummy string (+ 1 if sign is negative), put rlung-length of dummy string (-1 if sign is negative) in s starting at s[p2]. If sign is negative, put a '-' after the spaces (if any).
 After the sign, if any, put the dummy string.

If the length of s has changed, updated s[0].
 Add the number of characters written to s to p2.

Errors: If p2 < 1 or p2 > length of s + 1, set ioresult to istrovfl.
 If the write would extend the length of s past strmax(s), set ioresult to istrovfl.


```

procedure fwritestrpaoc(var s           : string;
                        var p2         : integer;
                        var a           : window;
                        aleng,
                        rleng           : shortint);

```

Purpose: To write (formatted) a packed array of characters into a string.

Parameters:

- s the string.
- p2 the index into the string. Initially where the write is to start. Finally one past the last character written.
- a the packed array of characters to be written.
- aleng the size of the packed array of characters (max 255).
- rleng the field width to be written into.

Stack:

```

----- <- sp + 22
| strmax(s)    |//////////|
----- <- sp + 20
| ptr to s     |
----- <- sp + 16
| ptr to p2    |
----- <- sp + 12
| ptr to a     |
----- <- sp + 8
| aleng        |
----- <- sp + 6
| rleng        |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Convert a into a string of length aleng. Call fwritestr with s, p2, the constructed string, and rleng.

Errors: Boundary errors may arise if aleng > 255. Fwritestr may set ioresult.

```

procedure fwritestrreal(var    r      : string;
                        var    p2     : integer;
                        x      : real;
                        w,
                        d      : shortint);

```

Purpose: To write (formatted) a real to a string.

Parameters: r the string.
p2 the index into the string. Initially where the write is to start. Finally one past the last character written.
x the real to be written.
w the field width to be written into.
d the number of digits after the decimal point.

Stack:

```

----- <- sp + 22
| strmax(r)  |//////////|
----- <- sp + 20
| ptr to r   |
----- <- sp + 16
| ptr to p2  |
----- <- sp + 12
| ptr to x   |
----- <- sp + 8
| w          |
----- <- sp + 6
| d          |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Convert the real (x) to a string representation right justified in a field width of w with d digits to right of the decimal point.

If it will fit, move this string representation into r starting at p2 and update the length of r if necessary.

Set p2 to one past the character written.

Errors: If p2 < 1 or p2 > length of s + 1, set iresult to istrovfl.
If the write would extend the length of s past strmax(s), set iresult to istrovfl.

```

procedure fwritestrstr(var s           : string;
                      var p2          : integer;
                      anyvar t        : string255;
                      rleng           : shortint);

```

Purpose: To write (formatted) a string into another string.

Parameters: s the string to be written into.
p2 the index into the string. Initially where the write is to start. Finally one past the last character written.
t the string to be written.
rleng the field width to be written into.

Stack:

```

----- <- sp + 20
| strmax(s)  |//////////|
----- <- sp + 18
| ptr to s   |
----- <- sp + 14
| ptr to p2  |
----- <- sp + 10
| ptr to t   |
----- <- sp + 6
| rleng      |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: If $rleng < 0$ then set $rleng$ to the length of t .
If $rleng >$ the length of t , replace $s[p2]$ to $s[p2+rleng-(length\ of\ t)-1]$ with spaces and add $rleng-(length\ of\ t)$ to $p2$.

Copy $rleng$ characters of t into s starting at $s[p2]$.
Add $rleng$ to $p2$.
If $p2 + length\ of\ t - 1 >$ length of s , set length of s to $p2 + length\ of\ t - 1$.

Errors: If $p2 < 1$ or $p2 >$ length of $s + 1$, set $ioresult$ to $istrovfl$.
If $p2 + rleng$ (adjusted) $- 1 >$ $strmax(s)$ (i.e. t won't fit into s starting at $s[p2]$), set $ioresult$ to $istrovfl$.

```

procedure fwritestrword(var s           : string;
                        var p2         : integer;
                        i,
                        rleng           : shortint);

```

Purpose: To write (formatted) a short (16 bit) integer to a string.

Parameters: s the string.
p2 the index into the string. Initially where the write is to start. Finally one past the last character written.
i the short integer to be written.
rleng the field width to be written into.

Stack:

```

----- <- sp + 18
| strmax(s)  |//////////|
----- <- sp + 16
| ptr to s   |
----- <- sp + 12
| ptr to p2  |
----- <- sp + 8
| i          |
----- <- sp + 6
| rleng      |
----- <- sp + 4
| return address |
----- <- stack pointer

```

Action: Call fwritestrword with s, p2, i, and rleng.
Note that i is passed by value so this is possible.

Errors: See fwritestrword.

```
procedure fwriteword(var t : text;
                    i,
                    rleng : shortint);
```

Purpose: To write (formatted) a short (16 bit) integer to a text file.

Parameters: t the text file.
i the short integer to be written.
rleng the field width to be written into (max 255).

Stack:

```
----- <- sp + 12
| ptr to t |
----- <- sp + 8
| i |
----- <- sp + 6
| rleng |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: Call fwriteint with t, i, and rleng.
Note that i is passed by reference so this is possible.

Errors: See fwriteint.

```

function scantitle (      fname      : fid;
                      var   fvid     : vid;
                      var   ftitle   : fid;
                      var   fsegs    : integer;
                      var   fkind    : filekind): boolean;

```

Purpose: Given a file identifier, to return the volume id, the rest of the file id (i.e. without volume id), the file size specifier, and the filekind associated with the suffix in the file id.

Parameters: fname the file identifier input.
 fvid the volume id returned.
 ftitle the rest of the file id returned.
 fsegs the file size specifier returned.
 fkind the filekind returned.

Stack:

```

----- <- sp + 26
| func. result |//////////|
----- <- sp + 24
| ptr to fname      |
----- <- sp + 20
| ptr to fvid      |
----- <- sp + 16
| ptr to ftitle    |
----- <- sp + 12
| ptr to fsegs     |
----- <- sp + 8
| ptr to fkind     |
----- <- sp + 4
| return address   |
----- <- stack pointer

```

Action: Call zapspaces with fname.
 If no file name is left return false.
 Otherwise, if there appears to be an SRM volume password immediately to the left of the colon (i.e. '<... max 16 chars ...>'), move it to the right of the colon.

Extract the volume id.
 If there is an illegal volume id then return false.
 (Legal volume ids include '#nn', '#nn:', ':', '*', '*:', and 'cccccccccccccc:'.)
 The colon (if any) is removed.
 If after removing the colon '*' is left, use the system volume id.
 If nothing is left, use the default (prefix) volume id.
 Set fvid to the volume id.

Extract the file size specifier.
If no legal size specifier is found, set fsegs to 0.
(Legal size specifiers include [*] and
[non-negative integer]).
If it is [*] then set fsegs to -1.
Otherwise, set fsegs to the non-negative integer.

Set ftitle to "all the rest" (i.e. the file id
without spaces and control characters, volume
identifier, colon, and file size specifier.)

Call suffix with ftitle and set fkind to the
filekind returned.

Errors: No file name left after call to zapspaces,
return false.
Illegal volume id, return false.

function suffix (var ftitle : string): filekind;

Purpose: To determine the filekind associated with the suffix of a filename.

Parameters: ftitle the filename.

Stack:

```
----- <- sp + 12
| function result          |
----- <- sp + 10
|strmax(ftitle)//////////|
----- <- sp + 8
| ptr to ftitle           |
----- <- sp + 4
| return address          |
----- <- stack pointer
```

Action: Search suffixtable from untypedfile to lastfkind.
If suffix in table entry matches suffix of ftitle (uppercased) then return the filekind index.
Stop when first match is found.
If no match is found, return suffix = datafile.

Errors: None.

procedure zapspaces (var s : string);

Purpose: To remove all spaces and control characters from a string.

Parameters: s any string.

Stack:

```
----- <- sp + 10
| strmax(s) |//////////|
----- <- sp + 8
| ptr to s |
----- <- sp + 4
| return address |
----- <- stack pointer
```

Action: String is scanned and all characters \leq ' ' or $=$ chr(del) are removed. (del = 127)

Errors: None.

Chapter 9

Directory Access Methods

Reference Specification for DAM's

This chapter describes what a Directory Access Method (DAM) must be able to do in order work properly with the File Support routines and Transfer Methods. A good way to get the most out of this text is to scan it, then go through it again while examining one of the DAMs supplied with the system. The LIF or WS1.0 DAMs are about equally good choices.

type

```
damrequesttype =
  (getvolumename, setvolumename, getvolumedate, setvolumedate,
  changename, purgename, openfile, createfile, closefile,
  purgefile, stretchit, makedirectory, crunch, opendirectory,
  closedirectory, catalog, makelink, setunitprefix, openvolume,
  duplicatelink, openparentdir, catpasswords, setpasswords,
  lockfile, unlockfile, openunit );

damtype = procedure
  (anyvar f:fib; UNUM:unitnum; request:damrequesttype);
```

Every DAM is a procedure taking three parameters. Usually the parameters take the following interpretation:

The first is a File Information Block for any type of file. The second is a unitnumber (an index into the Unitable). The third is a scalar telling what the caller wants the DAM to do. All other information the DAM uses can be found in the FIB or the Unitable.

Nota Bene: the first parameter F is an ANYVAR, which means the Compiler will accept anything that has an address (any variable; not a constant or expression). In a few cases something other than a FIB is passed, and the DAM internally coerces the argument's address into some other type. This may be confusing on first sight; instances of this behavior are pointed out in the commentary below.

It is best to implement each request as a procedure within the DAM, or anyway within the module containing the DAM. We will discuss each damrequesttype as if it were a separate routine.

The Golden Rule for DAMs

All of the following routines should (unless no access is necessary) verify that the volume name of the disc medium installed in the unit is the same as the UVID in the unit table. If not, the routine should make these changes in the unit entry for the volume:

1. Set UMEDIAVALID to false.
2. Set UVID to the correct name (the one actually found on the volume label), or to "" (the nil string) if no recognizable medium.

This rule is designed primarily for protection in the case of removable media on devices with no "door has been opened" state flag. It also guards against tricksters who manipulate the IO subsystem in unforeseen ways.

Calling DAM's

OPENFILE and CREATEFILE

On entry the DAM parameter F is actually a FIB with fields initialized by the File Support level. Some of the initialization is performed by SCANTITLE, a procedure which does preliminary parsing of file specifications.

- Both FUNIT and DAM parameter UNUM indicate the desired unit.
- FVID is the volume name derived from the original file name.
- FTITLE contains the original file name with these components removed:
 - Spaces and control characters (ord in [0..31,127])
 - Volume name and '?' (if any)
 - Size specification in brackets [<size>] (if any)
- FKIND reflects the suffix of the file name.
- FRECSIZE = 0 means the file is untyped (declared 'FILE').
- FISTEXTVAR = true means the file was declared type TEXT.
- FBUFFERED = true means FRECSIZE > 0.
- FEFT = the HP external file type.
- FREPTCNT = 0.
- FLASTPOS = -1.
- FPATHID = -1.

The following guidelines apply both to opening an old file and creating a new one:

- If no medium is present in the unit, set IORESULT to ord(inodirectory). You can tell this if the driver returns IORESULT = znomedium when it tries to access the disc.
- The volume name FVID need not necessarily be retained, but it probably should be since it is useful when closing or stretching a file.
- The file title FTID should be set to the "root" of the file name, by which is meant that part of the file specification which is not volume name, path id, passwords, file size, or other miscellaneous stuff. This root name is what the system uses to identify P-loaded programs. If FTITLE doesn't syntax correctly for this DAM, set IORESULT to ord(ibadtitle).

Certain FIB assignments are made in common by both the CREATEFILE and OPENFILE routines:

```

FPEOF :=      Physical end of file; this is the total number of
               bytes allocated for the file. Note the convention
               that the first byte of a file is byte zero, so FPEOF
               is the "index" of the byte after the last possible
               byte of data. If the DAM doesn't worry about the
               physical end of file (eg non-contiguous file systems),
               FPEOF should be set to maxint.

AM :=        Name of the Access Method procedure, chosen by
               whatever rules the DAM likes. The policy we like is:

               if unblocked device      use TM from the unit table
               else if not FBUFFERED    use amtable^[untypedfile]
               else                      use amtable^[FKIND]

FILEID :=    Some sort of identification appropriate for the
               Transfer Method; for simple file structures
               this is probably just the byte offset from the
               beginning of volume to the beginning of file.

PATHID      May be used any way the DAM well pleases.

```

OPENFILE

Opens an existing file. This involves finding the file by name in the directory, and setting fields of the FIB.

```

FKIND :=     Type of the file.

FISNEW :=    False.

FLEOF :=     Logical end of file; this is the number of bytes of
               valid data in the file. It indicates where EOF gets
               true, and where to start APPENDING if the file is
               extended. Same convention as FPEOF: first byte is
               number zero.

```

The opening operation may fail for any number of reasons: volume not present, storage medium switched, no such file, file already opened exclusively for someone else.

```
if open was successful then (*and only then*)
  if not UMEDIAVALID then
    begin
      close all OTHER open files (*especially temporary files*)
      UMEDIAVALID := true;
    end;
else (*open failed*)
  IORESULT := ord(inofile); (*or other appropriate value*)
```

CREATEFILE

Makes a new file. This involves allocating a directory name slot and initial space for the file, verifying that there isn't already a file of this name, and filling in the new directory entry.

If a file of the same name already exists, the preferred behavior is to ignore the old file until the new one is LOCKed, at which time the old one is purged. The create operation also has to deal with "anonymous" files, if the FIB so designates by the FANONYMOUS flag. Either of these situations may require generating a random file name since some DAM's unfortunately don't allow "temporary" files which could serve the needs of both anonymous and duplicated file names.

On entry to this routine, FIB field FPOS reflects the requested size of the file.

```
FPOS > 0      The file must be guaranteed at least FPOS bytes
              of available space.

FPOS = 0      No size was specified. This suggests the DAM
              should allocate as large a space as "possible",
              whatever that means. If the directory method
              uses contiguous files, the largest space is
              the biggest "hole" in the allocation map.

FPOS < 0      Indicates that the size specifier was '['*']'
              which suggests that about half the available space
              be allocated. In contiguous files, this is usually
              interpreted to mean the second largest space or half
              of the largest, whichever is greater.
```

The steps in creating a new file are given here.

```
if not UMEDIAVALID then
  begin
    close all OTHER open files (*especially temporary files*)
    UMEDIAVALID := true;
  end;

FISNEW := true;

FLEOF := 0; (*The file has no contents yet*)
```

Allocate the space and directory entry. If unsuccessful, IORESULT should be set to:

```
ord(inoroom)   if out of data space on volume.  
ord(idirfull)  if no space in directory.  
ord(idupfile)  if there is another of the same name and the DAM  
                can't cope.
```

PURGEFILE

The parameter F is actually a FIB. The purpose of the call is to purge the physical file associated with the FIB (said physical file must be open). This is a little different from purging a file by name, since some DAM's may allow more than one file to be (temporarily) open under any particular name (FTID).

Verify that FVID (the volume name in the FIB) matches the volume name on the medium. If no, set IORESULT to ord(ilstfile).

If the volume is right but the file name can't be found (eg if someone else purged it), set IORESULT to ord(ilstfile). Some DAMs may remember how many logical files are open to a given physical file and refuse to purge if the file is in use.

The physical file must now be closed, although for many implementations this requires no special action. Then the physical file is destroyed, that is, deleted from the volume directory.

Fields FISNEW, FVID, FTID, PATHID, FUNIT and FILEID are still valid, retaining the values placed there by OPENFILE or CREATEFILE.

CLOSEFILE

Parameter F is actually a FIB, and the physical file associated with F must be open.

Verify that FVID (the volume name in the FIB) matches the volume name on the medium. If no, set IORESULT to ord(ilstfile).

If the volume is right but the file name can't be found (eg if someone else purged it), set IORESULT to ord(ilstfile). Some DAMs may remember how many logical files are open to a given physical file and refuse to purge if the file is in use.

The value of FLEOF is the final end-of-file which will be recorded in the volume directory. FPEOF retains the physical limit value assigned by OPENFILE, CREATEFILE, or STRETCH. FISNEW, FVID, FTID, PATHID, FUNIT and FILEID are still valid. FMODIFIED indicates whether FLEOF or FPEOF have been changed; many DAM implementations will not need to take any action with respect to the volume directory if the size of the file hasn't changed.

NB: SEEK does not change either FLEOF or FPEOF; to force a file to be extended, you need to write something! Neither will altering a record within the size limit currently specified by FLEOF cause FMODIFIED to be set true.

STRETCHIT

The purpose of this routine is to extend the physical limit of a file. Not all DAMs can necessarily do this. For instance, the UCSD DAM can only stretch a file if there happens to be free space after it on the medium. Generally you can assume

- If the directory distinguishes between logical and physical eof, it ought to be possible at least to extend the file's logical eof up to the limit of the physical eof. Not all directories retain enough data to make this distinction; sometimes FPEOF := FLEOF when the file is closed.
- If the volume space is managed on the basis of contiguous files, a file should be stretchable if there is free space after it.
- If disc space is allocated in a non-contiguous way, such as using extents, linked lists or tree structures, files probably can be stretched until the volume is full.

For calls to STRETCHIT, the parameter F is actually a FIB, and the physical file associated with F must be open. FPOS contains the desired size of the file in bytes (not the amount to stretch). It is usually desirable to allocate a "reasonably large" amount of additional space to the file, since adding a minimal amount will probably result in repeated calls to stretch, adversely affecting performance.

If the stretch succeeds (there is enough space), set FPEOF to the new physical end of file and set FMODIFIED to true. Note that STRETCHIT does not return an error if it fails; instead it leaves FPEOF unchanged. To tell if the stretch succeeded, the caller must compare the requested FPOS to FPEOF after the stretch.

GETVOLUMENAME and SETVOLUMENAME

The actual parameter F is a string variable at least 16 bytes long, instead of a FIB. The routines read or write, respectively, the name of the medium currently mounted in the unit selected by the DAM parameter UNUM.

If the unit is an unblocked device, the name in uvid of the unit entry is returned. If there is no recognizable medium, the nil string " should be returned.

GETVOLUMEDATE and SETVOLUMEDATE

The actual parameter F is a variable of type datetimerec instead of an FIB. datetimerec is exported from SYSGLOBALS.

```

type daterec = packed record
    year: 0..100;
    day: 0..31;
    month: 0..12;
end;
timerec = packed record
    hour: 0..23;
    minute: 0..59;
    centisecond: 0..5999;
end;
datetimerec = packed record
    date: daterec;
    time: timerec;
end;

```

When month = 0 the date is invalid; some file systems may use year = 100 in the creation date to denote temporary files.

These routines read or write, respectively, the date and time associated with the volume label of the medium currently installed in unit UNUM. This generally is the creation date of the volume. If the operation is not applicable to the directory format, the value of the daterec parameter should be unchanged.

CRUNCH

This operation is useful for disc formats using contiguous files. Its purpose is to move files on the volume as necessary so that all free space is contiguous. The operation is "silent", ie it doesn't report on its progress to the CRT as was the case in the 1.0 release.

On entry, F is an FIB containing FVID, FUNIT and FTITLE as in OPENVOLUME. There must be no open files in the volume! It is especially important that the crunch implementation verify that the right volume is installed in the unit.

PURGENAME

Remove a permanent file from the directory by name. Parameter F is a FIB containing FVID, FUNIT and FTITLE as in OPENFILE. Note the clear separation between FS and DAM operations; this operation of purging a file from a directory has nothing to do with closing the logical file (the FIB)! If no such file is found, set IORESULT = ord(inofile).

CHANGENAME

Change the name of a file in the directory. F is a FIB with FVID, FUNIT and FTITLE as in OPENFILE. FWINDOW is a pointer to the new desired name, which is a string no longer than file names are permitted to be by the DAM. If the file name doesn't parse properly for this DAM, set IORESULT = ord(ibadtitle). If the change would duplicate an existing permanent file name, set IORESULT = ord(idupfile).

MAKEDIRECTORY

Create a new directory containing no files. This corresponds to the Filer's Zero operation. On entry, **F** is actually a **FIB** containing **FVID**, **FUNIT**, and **FTITLE** as in **OPENFILE**. **FWINDOW** is a pointer to a **catentry**, the fields of which are as follows:

- **CNAME** is the desired name of the new directory.
- **CEXTRA1** is the number of file entries desired; zero passed in allows the **DAM** to decide default number.
- **CPSIZE** is the total available physical space on the unit.

The type **catentry** is exported by kernel module **MISC**. It is a general structure able to describe many possible variations on file naming.

MAKEDIRECTORY is expected to do its thing unless absolutely impossible (eg no medium mounted in unit), in which case it should come back with the appropriate **IORESULT**.

OPENDIRECTORY and OPENPARENTDIR

Since directories may be arbitrarily long, they are dealt with by scanning through them in a series of sequential operations. Regardless of the actual directory structure, this serializing operation presents the directory as if it were an array of directory entries, a few of which can be examined at a time. Obviously the way to manipulate a directory is to use an **FIB** to describe it. If you look in the **DAMs**, you will find that for simple sequential directory structures we have implemented the directories themselves as random-access files of directory entries.

OPENDIRECTORY is the first such operation; it gets information about a directory, and also prepares the **FIB** and directory for a cataloguing operation. In the case of the **SRM**, **OPENDIRECTORY AND OPENPARENTDIR** get a **pathid** for subsequent calls to **catalog**, **openfile**, **createfile**, **opendirectory**, **openparentdir** and so forth.

On entry, **F** is a **FIB** containing **FVID**, **FUNIT** and **FTITLE** as in **OPENFILE**. **FWINDOW** points to a **catentry** (not a directory entry of the type supported by the **DAM**, but our generalized catalog descriptor type).

On exit, **FTITLE** is the file name part of the original **FTITLE**. The fields of the **catentry** reached through **FWINDOW** contain this information:

- **CNAME** is the name of the directory.
- **CEXTRA1** is the maximum number of entries this directory could ever hold.
- **CPSIZE** is the physical size of the medium.
- **CLSIZE** is the size of the data portion of the medium.
- **CEXTRA2** is the unused space available.
- **CSTART** is the first legal (volume-relative) byte address for the data portion of a file.
- **CBLOCKSIZE** is the number of bytes in one sector or block.

- CCREATEDATE, CCREATETIME are the day and time the directory was created.
- CLASTDATE, CLASTTIME are when the directory was last modified.
- CINFO may contain other useful messages.

Various ioreults may be returned: inodirectory if the volume has no directory or a directory for some other DAM; ilostunit if the volume name doesn't match the unit table.

CATALOG

OPENDIRECTORY must be called first; then calls to CATALOG bring in sections of the directory. The parameter F to the CATALOG call must be the same undisturbed FIB returned by OPENDIRECTORY, except:

- FPEOF is the number of files on which the caller is requesting information.
- FWINDOW now points to an array [0..FPEOF-1] of catentry.
- FPOS is the "index" of the first file for which information is being requested; an FPOS of zero corresponds to the first file in the directory. Stated differently, element zero in the array of catentries to be returned will describe file number FPOS (indexing from zero) in the directory.

On exit, FPEOF is the actual number of files catalogued; it will be no greater than the number requested, but may be smaller. FPOS should remain the same. Elements zero through (FPEOF-1) of the array of catentry are filled in as follows:

- CNAME is the name of the file.
- CKIND is the file kind (codefile, textfile etc.)
- CEFT is the external file type (16-bit LIF code, for example)
- CPSIZE is the physical size (in bytes) of the file.
- CLSIZE is the current logical file size (in bytes).
- CSTART is the starting location (byte offset) of the file in the volume; it may be some other form of identification, generally corresponding to the FILEID field of an FIB.
- CBLOCKSIZE is the size in bytes of a sector or block.
- CCREATEDATE, CCREATETIME are when the file was created.
- CLASTDATE, CLASTTIME are when the file was last modified.
- CEXTRA1 and CEXTRA2 are additional implementation-dependent information; we use CEXTRA1 for the LIF "extension word".
- CINFO may contain further implementation-dependent messages.

CLOSEDIRECTORY

Terminate the association of the FIB which was set up by **OPENDIRECTORY** or **OPENPARENTDIR**. In many DAMs there will be nothing to do for this operation.

MAKELINK

This operation is provided only for hierarchical directories. Its purpose is to create a new access path (link) to a file from a directory which is not the original parent of the file. On entry, **F** is actually a FIB containing **FVID**, **FUNIT** and **FTITLE** as in **OPENFILE**; and **FWINDOW** is a pointer to the desired new path name.

SETUNITPREFIX

Set the default subdirectory or path name for a unit. It is possible to have several workstation logical units which are connected to Shared Resource Managers (or even to the same SRM). Each of these units can have a "default" pathname, designating the current working directory for the unit. What this means is:

- If the pathname to a file begins with a slash '/', the path will be followed down from the root.
- If the pathname is absent or does not begin with a slash, the path will be taken as starting from the current working directory.

On entry to this DAM call, **F** is an FIB containing **FVID**, **FUNIT** and **FTITLE** as in **OPENFILE**.

OPENVOLUME

Open a whole volume (unit) as a single file. The interface is like **OPENFILE**, but **FLEOF** and **FPEOF** reflect the physical size of the volume, and no temporary file cleanup is done. This function is useful for such purposes as complete volume transfers during a backup operation.

It is not useful for the SRM DAM; for the LIF and UCSD DAMs, it is translated into a call on the "unblocked DAM" exported from MISC. The unblocked DAM is a very limited implementation, which sets up just enough FIB information to allow a simple byte-stream Access Method to access the volume directly.

The LIF Directory Access Method

This section examines the LIF DAM as an example to help you understand how DAMs work. The Pascal 2.0 LIF DAM is a superset of the LIF specification, allowing access to 9826 BASIC files and providing compatibility with 9826 Pascal release 1.0 file naming conventions. The extensions are only in the allowable names for files -- Pascal 2.0 is less restrictive than the Standard.

The LIF DAM is written as a program with the following overall structure:

```
program instlifdam;

  module lifmodule;
  imports ...
  exports
    procedure lifdam ( ... );
    procedure installlifdam;
  implement
    ...
  end; {lifmodule}

begin {program instlifdam}
  installlifdam;
end.
```

The intent is that program INSTLIFDAM be compiled and placed in INITLIB along with LIFMODULE. During bootup, INSTLIFDAM is permanently loaded along with the rest of INITLIB, and then it is executed. Its execution simply results in calling procedure INSTALLLIFDAM, which leaves things in such a state that LIFDAM can be called when needed. Recall that DAMs are called as procedure variables, through a Unitable entry whose value assigned by the execution of the configuration program TABLE.

Procedure INSTALLLIFDAM is very simple. It merely allocates from the heap a file variable called DIR, which is hidden within the module implement part. DIR is a file of LIF directory entries, which is used to read the disc directory. (This recursive trick should not be too confusing. The LIF directory simply looks like a contiguous sequence of directory entries, so why not read it as a file?)

Most of the remainder of this discussion pertains to procedure LIFDAM, which does the dirty work. LIFDAM is structured so that the procedures which implement each type of DAM request are nested within it:

```

procedure lifdam
  (anyvar f:file;           {file descriptor}
   unum:unitnum;          {logical unit number}
   request:damrequest:type); {requested action}

  declarations of:
    local types
    utility routines
    procedures to implement various DAM requests

begin {body of lifdam}
  lockup;      {stop key interlock}
  disable media change error reporting;
  set up DIR file to reference desired unit;
  ioresult := no error;
  try
    case request of
      openfile:
      createfile:
      :
      :   various DAM requests, mostly procedure calls
      :
    end;
  recover
    if (escapecode<0) {system escape code}
       and {escapcode<>-10} {not IO error}
    then
      begin lockdown; escape(escapecode) end;
    enable media change error reporting;
    lockdown; {release interlock}
  end; {lifdam}

```

The LOCKDOWN and LOCKUP operations, provided by module KBD, serve to keep the STOP key from interrupting critical operations such as rewriting a disc directory. The mode of operation is that a record is kept of certain types of requests made through the keyboard interrupt routine, and these are executed at a later (presumably safe) time. Locking is discussed in the section on keyboard handling.

Implementation of LIFMODULE

The module implementation begins by defining some types and utility routines. Type VNAME is a 6-character array representing the name of a LIF volume. LVHEADER describes the so-called "volume header" at the front of every LIF volume. This header names the volume and specifies certain characteristics of the disc medium such as tracks per surface and sectors per track. It also indicates the date the volume was created. LIFNAME is a 10 character packed array which gives the external name of a file.

Each LIF directory entry has this form:

```
direntry = packed record
    fname: lifname;
    ftype: integer16;    {16-bit integer}
    fstart: integer;
    fsize: integer;
    fdate: tdate;
    lastvol: boolean;
    volnumber: word15;  {15-bit unsigned}
    extension: integer;
end;
```

LIF allows for a file to span multiple volumes, which is the function of fields VOLNUMBER and LASTVOL. The multi-volume capability is not implemented by the Pascal 2.0 DAM.

The EXTENSION field deserves special mention. This field is in some sense a little extra data which goes with a LIF file. It can generally be used however the file system wants, but for certain file types such as LIF ASCII (Pascal .ASC suffix) the extension must be zero. For data files (file of <type>), Pascal uses the extension to retain the logical (as opposed to physical) end-of-file. There is more information about this with the description of the FSTARTADDRESS field of the FIB.

The fields FSTART and FSIZE are sector numbers on the disc (an HP disc sector is 256 bytes). FSIZE is the number of sectors allocated to the file.

Type SPACEREC is used to record information about free "holes" in the disc's allocation structure. CATENTRY, which is referenced by this DAM, is a type exported from FS; you may recall that it is a sort of "normalized" directory entry. Whenever directory entries are read by a DAM and passed back to the caller, they are transformed from the form native to the DAM into the canonical representation of a CATENTRY for processing in a standard way by the file system.

We already mentioned DIR, which is a pointer to a file of LIF DIRENTRYs.

Next after the types local to LIFDAM are declared some utility routines which do such things as deblanking a LIF name, converting a date and time in system format to the format expected by LIF directory entries, and handling file name suffixes.

LIF Directory File Names

The LIF Directory Access Method generally allows any ASCII character to be used in a file name. This is contrary to the HP LIF Standard, which states that file names must be composed only of upper-case letters, digits, and the underscore '_' character. Note that upper and lower case letters are distinct. File names stored in LIF directories are always exactly 10 characters; they may be blank-padded by the DAM if necessary.

The LIF DAM accepts only uppercase suffixes!

The 10-character file name length would be a very severe restriction when four or five characters are required for a suffix. To ease this problem, the LIF DAM performs a transformation on the file name which compresses the suffix if one is present. The transformation occurs automatically when a LIF directory entry is made, and it is reversed automatically before the file name is ever presented to any program or to the user.

This "black magic" is usually completely transparent to the Pascal user, although its effects may be seen when a LIF directory is examined from the BASIC language system. It sounds complicated and dangerous, but in practice it is very smooth. Most people would never notice it if they weren't told.

Here is how LIF DAM changes a name before putting it into the directory:

1. Look for a standard suffix. If there is none, the file is a data file and the name is used unchanged unless it is too long. If it is longer than 10 characters an error is generated.
2. If a suffix is found, it is removed from the name but the dot '.' delimiter is left. If the resulting name is longer than 10 characters, an error is generated.
3. If the trimmed name is not too long, the dot is replaced by the first letter of the suffix, eg 'A' for '.ASC'.
4. If the name is now less than 10 characters long, it is extended by appending underscores '_' to 10 characters.

Using this algorithm, we would have the following examples:

```
'A.ASC'           ==>  'AA_____'  
'charlie'        ==>  'charlie  '  
'123456789.TEXT' ==>  '123456789T'  
'GollyGeeeT'     ==>  rejected because it would be  
                      confused with transformation of  
                      'GollyGeee.TEXT'
```

The reverse transformation is fairly obvious:

1. If the 10th character is a blank, do nothing; otherwise,
2. Remove all trailing underscores.
3. Compare the last non-underscore to the first letter of each valid suffix. If a match is found, remove that letter from the file name and append a dot '.' followed by the full suffix.
4. If no suffix match is found, use the original file name.

Routines within Procedure LIFDAM

Parameter F is an FIB, or occasionally another record type depending on the value of REQUEST. UNUM is the unit number of the device on which the requested operations are to be performed. REQUEST specifies the operation to be performed.

Most operations have a secondary effect of checking the volume name in the unit table against the actual volume name. If the two do not match then the one in the unit table (UVID) is changed and the medium is marked as changed (UMEDIAVALID:=FALSE).

Initial actions: save current UMEDIAVALID bit from UNITABLE^[UNUM]; set UMEDIAVALID:=TRUE; clear UREPORTCHANGE; copy unit number into directory FIB DIR^; IORESULT:=INOERROR; set ANYCHANGE:=FALSE. On normal exit from the call, a true value in ANYCHANGE will cause the directory buffer to be flushed.

The remaining interesting routines are all declared within procedure LIFDAM. They operate on a few variables which are local to LIFDAM (but "global" to the procedures within it, by the normal scoping rules of Pascal). These variables are VOL, the header of the volume in question; VOLID, its system name; DENTRY, a directory entry; and DINDEX, DLAST, DEND which are integers used when scanning through the directory.

function LIFVOL

Uses the Transfer Method (specified in the Unitable) and FIB DIR^ to read the volume header from a LIF volume and verify that it does indeed seem to be in LIF format. If the volume name is not the same as the "expected" name found in the Unitable entry, LIFVOL sets umediavalid := false to indicate that the disc has been changed. After accessing the disc to clear any hardware medium change indication, reporting of medium change errors is enabled by setting UREPORTCHANGE.

A LIF volume name of six blanks will be rejected as invalid.

procedure OPENDIR

Calls LIFVOL for verification, then sets up the DIR^ FIB to look like an open direct-access file which can be used to read the LIF directory entries. Once this is done, operations like READ and SEEK can be used on the directory in the normal Pascal style. OPENDIR also sets DLAST to the number of directory entries and DEND to a value one greater than the number of the last directory sector on the disc.

Be aware that the LIF volume is not necessarily at the front of the disc! The location where the volume begins is specified by field BYTEOFFSET in the Unitable entry for the volume. The

Transfer Method takes care of translating volume-relative byte offsets (such as the physical and logical EOF indicators) into physical addresses on the disc.

OPENDIR finishes by reading the first directory entry.

procedure FLUSHDIR

Issues a "flush" request to empty the buffer associated with DIR^ when a directory is being rewritten;

procedure GETSYSDATE

procedure SETSYSDATE

procedure CVTDATETIME

Transform the date between LIF representation and the standard Workstation DATEREC type.

procedure CRUNCHV

Repacks the volume so that all files are contiguous and all the free space is at the end of the volume. Note that when LIF files are purged, they are not really lost; instead their type is changed to mark them purged. CRUNCHV will reclaim all the space held by purged files.

Moving is performed one file at a time, using as much memory as is available for temporary buffering. The directory is updated after each file is moved.

procedure DOMAKEDIRECTORY

Creates an empty directory on a LIF volume, and uses DIR^ to write it out.

procedure DOPENDIRECTORY

Implements the OPENDIRECTORY DAM request. Calls OPENDIR and CHECKVOLID, then sets up a CATENTRY to describe the volume which was opened.

procedure DOCAT

Implements the CATALOG DAM request. Recall that the purpose of this request is to read a specified number of directory entries into an array of canonical CATENTRYS.

procedure FINDFILE

Searches the directory for a file by name. The parameter TEMPORARY specifies whether the search is for a temporary file or a permanent one. Temporary files are denoted by a month of creation set to 99. Note that anonymous files are identified by the fact that they start at the "correct" place on the volume; the FIB field FILEID is simply the offset of the first sector of the file.

procedure PURGEF

Purges the file described by DENTRY. This is done by setting its LIF type to zero and rewriting the directory entry.

procedure DOPURGENAME

Implements the corresponding DAM request. Calls FINDFILE and PURGEF to do the work.

procedure CLEANDIR

Removes all the temporary files from the directory. This is necessary when, for instance, a program aborts or the medium in a disc drive has been changed.

function GETSPACE

Implements the space allocation policies for LIF. The amount of space requested is determined by the parameter SPACE:

- **SPACE > 0** : The request is for a specific amount of space, which will be rounded up to the nearest sector (multiple of 256 bytes).
- **SPACE = 0** : Request for largest available block of free space.
- **SPACE < 0** : Request for "about half the free space", which is the greater of (half the largest hole) or (the second largest free hole).

The value of SPACE is determined by the File Support level, according to an analysis of the name of the file. If the file name ends in the characters '[*]', the value of SPACE will be minus one ("about half"). If the file name ends in '[nnn]', the integer nnn is multiplied by 512 to determine the value of SPACE. This is a request for 512*nnn bytes. The use of 512 byte blocks rather than bytes is a historical hangover. If there is no bracket-notation, the value of SPACE will be zero ("the largest free block").

To understand GETSPACE, you need to know how the LIF organization works. It is quite a bit more complicated than one might initially expect.

The LIF directory is usually placed at the front of the disc volume, right after the header; but it need not be there. The header has a field called DSTART which tells where the directory begins. The LIF DAM habitually places Pascal LIF directories right after the header, which is 2 sectors long. Another field, DSIZE, tells the number of sectors allocated for the directory. Each sector holds up to 8 directory entries.

Files with FTYPE = 0 are purged files. The last directory entry has FTYPE = -1. The fundamental rule of LIF is that files in the directory must appear in the same order as files in the volume. If the directory is completely full, the -1 entry will not be present and DLAST is used to determine when the end of directory has been reached.

Free space in LIF may appear in two ways: a "hole" between the spaces defined by two directory entries, or the space past the last file, which has never been allocated. Note that a file with FTYPE = 0 is not a reliable way to measure open space; the file may have been broken up and part of it given away.

You may wonder how a hole can be created between two directory entries, since a file still keeps its slot in the directory after being purged. The answer is that after a file has been purged, part of its space may be re-allocated. When this happens, the purged directory entry is re-used to denote the new file, and the "hole" which remains can only be detected as the difference between the end of one file and the start of the next.

When reclaiming purged files, GETSPACE must also be able to combine adjacent purged files into one big free area. This happens as a side-effect of the process of scanning for free space.

Finally, note that directory entries for purged files must also be managed. For instance, suppose the directory is full but a big chunk of free space exists at the distal end of the volume. Suppose further that the third directory entry describes a purged file. If a large space request is issued, which can only be satisfied by the chunk at the end, the third directory entry must be re-used. Since LIF requires that directory entries appear in the same order as the files appear on the volume, all the directory entries must be "scooted left" to open up a free entry at the end of the directory. In general, a fully capable LIF handler must be prepared to shuffle directory entries either left or right. Moreover, because the entries can move, a correctly implemented LIF DAM can never depend on the physical location of a directory entry; it must always search!

Not all LIF implementations are nearly this fancy. Many just give up in disgust at the slightest difficulty in getting either space or a free directory entry. For this reason, the Pascal DAM may succeed in allocating file space in situations where BASIC or HPL will fail. Still, if the allocation succeeds, the resulting directory is valid and will be recognized by other LIF file systems.

With this background, you should be able to understand GETSPACE. The body of the procedure simply scans all the directory entries, looking for free space and determining if the directory is full. CHECKENTRY is called to process each entry; it is responsible for noticing adjacent free spaces. It also tries to manage things so that if a free hole is selected, the most convenient directory entry is allocated to that hole. To get its work done, CHECKSPACE is called to look at the space, ALLOCATE fills in fields in SPACEREC, and SHUFFLE is called to shuffle directory entries as required.

procedure FINISHFIB

Called from various places to "finish" setting up the user's FIB in response to various DAM requests. Also provides the service of selecting the Access Method for the file, based on the FKIND field of the user's FIB.

procedure OPENNEW

Called from OPENF to open a "new" file (which requires space to be allocated). Updates the directory to indicate that the new file has been opened. New files are always temporary until closed with LOCK.

procedure OPENOLD

Called from OPENF to finish opening an existing file.

procedure OPENF

Opens either an old or a new file, depending on the value of the FISNEW field of the user's FIB. Calls OPENNEW, OPENOLD, and FINDFILE as required.

procedure CLOSEF

Closes an open file. If the file is a new (temporary) file, then any old file of the same name will be purged first. If the file has been modified in such a way as to invalidate its directory entry (for example, if the file has been extended), then the directory entry is updated.

procedure STRETCHF

Tries to extend the file by an amount indicated by FPOS in the user's FIB. The mechanism for requesting is discussed in the section about FIB's; what happens is that FPOS is set to the desired new limit, and a STRETCHIT DAM request is issued.

STRETCHF first sees if the requested limit is beyond the current limit. If so, the file can only be stretched if a hole exists beyond the current physical end.

procedure CHANGEFNAME

Implements the CHANGENAME DAM request. Searches for an existing file and either rewrites its directory entry (thus changing the name) or returns an IORESULT <> 0.

procedure DOOVERWRITEFILE

Implements the OVERWRITEFILE DAM request. If the file exists, calls OPENOLD then changes the FIB and directory entry to show it as a new temporary file. If the file doesn't exist then OPENNEW is called.

procedure NOWOPEN

First makes sure the disc hasn't been changed, then makes sure the file exists.

This concludes the list of support routines in procedure LIFDAM. As explained in the beginning of this section, LIFDAM selects the correct routines by a case statement which branches on the type of DAM request.

Details on Various DAM Requests

The notes which follow detail the setup and state required for various DAM requests processed by LIF DAM. If you try to implement a new DAM, it would be a good idea to outline your setup conditions and actions in a fashion similar to this.

OPENFILE

Set values in the FIB F to allow read/write operations to be done to the file identified in the FIB.

```
FIB entry setup
  fistextvar
  fbuffered
  fanonymous (must be false)
  fvid
  funit
  ftitle

FIB exit changes
  fisnew := false
  ftid   := ftitle
  fkind
  fileid := start address(volume relative in bytes) of file
  fpeof  := size of file in bytes
  fleof  := logical size of the file in bytes
  fmodified := false
  fstartaddress
  feft
  am      determinded by fbuffered, fistextvar and fkind
```

As a secondary function, if UMEDIAVALID is false then all temporary files are purged from the directory AFTER the file is successfully opened.

```
UMEDIAVALID := true;
  (It may happen that header validation will set umediavalid
  false and open will set it true in the same call to the DAM.)
```

CREATEFILE

Allocate space on the volume for a temporary file; set values in the FIB (F) to allow read/write operations to be done to this file.

```
FIB entry setup
  fistextvar
  fbuffered
  fanonymous
  fkind
  feft
  fpos      size of file in bytes or 0 or negative
  fstartaddress
```

```
fvid
funit
ftitle
```

FIB exit changes

```
fisnew := true
ftid   := ftitle (for non anonymous files only)
fileid := start address(volume relative in bytes) of file
fleof  := 0
fpeof  := allocated size of the file in bytes
fmodified := true
am      determind by fbuffered , fistextvar and fkind
```

As a secondary function, if UMEDIAVALID is false then all temporary files are purged from the directory BEFORE the file is created.

```
UMEDIAVALID := true;
(It may happen that header validation will set umediavalid false
and the create will set it true in the same call to the DAM.)
```

OVERWRITEFILE If the file identified in FIB F exists then use its space instead of using the space allocation routines. Then operate as for CREATEFILE. If the file does not exist then the operation performed is CREATEFILE.

FIB entry setup

```
fistextvar
fbuffered
fanonymous must be false
fkind
left
fpos      size of file in bytes or 0 or negative
          (used only if the file does not exist)
fstartaddress
fvid
funit
ftitle
```

FIB exit changes

```
fisnew := true
ftid   := ftitle
fileid := start address(volume relative in bytes) of file
fleof  := 0
fpeof  := allocated size of the file in bytes
fmodified := true
am      determind by fbuffered , fistextvar and fkind
```

As a secondary function, if UMEDIAVALID is false then all temporary files are purged from the directory BEFORE the file is created.

UMEDIAVALID := true;

(It may happen that header validation will set umediavalid false and the create will set it true in the same call to the DAM.)

CLOSEFILE F identifies a currently open file. If this file is a temporary file (as created by **CREATEFILE** or **OVERWRITEFILE**) then if a permanent file of the same name exists, purge it. Otherwise mark this file permanent.

FIB entry setup

fvid

funit

fmodified if fmodified is false, this call is a no op.

fisnew

feft

fanonymous

ftid

FIB exit changes

no changes

PURGEFILE

F identifies a currently open file; mark it purged.

FIB entry setup

fvid

funit

fisnew

feft

fanonymous

ftid

FIB exit changes

no changes

STRETCHIT

F identifies a currently open file. If possible, extend the allocated space for the file.

FIB entry setup

fvid

funit

fisnew

feft

fanonymous

ftid
fpos the requested SIZE of the file in bytes

FIB exit changes

fpeof the new size of the file (this will not change if
 the request could not be performed and it may be
 larger than the requested size)

CHANGENAME

Fwindow (in F) points to a filename. Find the file identified in ftitle then change its name to the value in fwindow^.

FIB setup

funit
fvid
fwindow points to an fid (the new name);
fanonymous false;

FIB exit changes

no changes

GETVOLUMENAME

F is a string; place the volume name in this string.

SETVOLUMENAME

F is a string; replace the volume name with the contents of this string.

PURGENAME

F identifies a file, mark this file purged.

GETVOLUMEDATE

F is a daterecord, place the systemdate in this record. (Systemdate is a special datetime field near the end of the volume header sector.)

SETVOLUMEDATE

F is a daterecord; copy the value in this record to the systemdate field of the volume header. (Systemdate is a special datetime field near the end of the volume header sector)

CRUNCH

Move the directory entries and data area of the volume so as to leave all unused space at the end of the directory and volume.

```
FIB setup
  funit
  fvid
  ftitle      must be nil string
```

```
FIB exit changes
  no changes
```

CATALOG

Return an array containing information about the files in the directory excluding temporaries. fpos <= 0 indicates a request for the first file.

For a series of calls, set fpos to 0 and fpeof to the number of entries in the callers array. After the first call, set fpos := fpos + fpeof to CAT consecutive files.

```
FIB setup
  funit
  fvid
  fwindow      points to an array of CATENTRY 's
  fpos         index of first file to cat
  fpeof        number of entries to cat
```

```
FIB exit changes
  fpeof        the actual number of entries filled.
```

OPENDIRECTORY

Returns information about the directory (same format as catalog)

```
FIB setup
  funit
  fvid
  fwindow      points to a variable of type CATENTRY
```

```
FIB exit changes
  The array pointed to by fwindow is filled, otherwise
  no changes are made to the FIB.
```

CLOSEDIRECTORY

This is a no op; the volume header is NOT checked.

OPENVOLUME and OPENUNIT

Passes this request to UNBLOCKEDDAM; the volume header is NOT checked.

SETUNITPREFIX

Checks ftitle, it must be zero length. No other fields are checked, no changes to the FIB are made. The volume header is NOT checked.

All other DAM requests will return: IORESULT = IBADREQUEST.

Chapter 10

File Operations

Introduction

This chapter outlines typical file operations. A module containing many of these operations has been listed. After each procedure or function, a short commentary is provided. The last few sections of this chapter deal with writing your own Command Interpreter.

The listed module is named: **FILEPACK** and is a collection of sample procedures to perform common file operations.

The procedures in this module may be called from a program to perform the following operations:

- copy a file
- translate a file
- duplicate a link to a file
- change a file name
- list file passwords
- change file passwords
- remove a file
- make a file
- catalog a directory
- make a directory
- create a volume directory
- repack a volume
- list volumes on line
- set default and unit prefixes

FILEPACK is provided as a set of examples or guidelines to illustrate the use of the lowest level of the Pascal 2.0 file support system, particularly certain service requests to the Directory Access Methods (DAM's).

Filepack Examples

What follows is a commentary on this set of examples, which should help the reader to understand the requirements and calling sequences of the DAM's well enough to fashion similar code.

```
$sysprog$

module filepack;

import sysglobals,
    misc,
    fs,
    asm;

export
    type volumearray = array[1..50] of string[25];

    procedure volumes (var v: volumearray);
    procedure filecopy (filename1,
        filename2: fid; format, writeover: boolean);
    procedure duplicate(filename1,
        filename2: fid; purgeold: boolean);
    procedure change (filename1,
        filename2: fid);
    procedure repack (filename: fid);
    procedure createdir(filename: fid; newname: vid; entries, bytes: integer);
    procedure makefile (filename: fid);
    procedure makedir (filename: fid);
    procedure remove (filename: fid);
    procedure prefix (filename: fid; unitonly, sysvol: boolean);

    procedure startcat(filename: fid;
        var dirname: vid;
        var typeinfo: string;
        var createdate, changedate: daterec;
        var createtime, changetime: timerec;
        var blocksize, phy_size, start_byte, free_bytes, max_files: integer);
    procedure cat(filename: integer;
        var filename: tid;
        var typeinfo: string;
        var createdate, changedate: daterec;
        var createtime, changetime: timerec;
        var kind: filekind;
        var eft: shortint;
        var blocksize, logical_size,
            phy_size, start_byte,
            extension1, extension2: integer);
    procedure endcat;

    procedure startlistpass(filename: fid);
    procedure listattribute(wordnumber: integer; var outstring: string);
    procedure listpassword(wordnumber: integer; var outstring: string);
    procedure changepassword(word: passtype; attrlist: string255);
    procedure endpass;
```

```

function ioerrmsg(var msg: string): boolean;

implement

const
    catlimit      = 200;

type
    buftype      = packed array[0..maxint] of char;
    bigptr       = ^buftype;
    closecode    = (keepit, purgeit);

    catarray     = array[0..catlimit] of catentry;

    passarray    = array[0..catlimit] of passentry;
    passarrayptr = ^passarray;

var catfib, passfib: ^fib;
    catentptr: ^catarray;
    wordlist, optionlist: passarrayptr;

```

```
function memavail $alias 'asm_memavail'$ :integer; external;
```

FUNCTION MEMAVAIL is a system routine which returns the number of bytes of free memory, that is, the space between the stack pointer and the heap pointer.

```
function min(a, b: integer): integer;
begin if a < b then min := a else min := b end;
```

FUNCTION MIN returns the lesser of two integers.

```
function ioerrmsg(var msg: string): boolean;
begin
    if ioresult=ord(inoerror) then ioerrmsg := false
    else
        begin
            ioerrmsg := true;
            getioerrmsg(msg, ioresult);
        end;
end;    { ioerrmsg }
```

FUNCTION IOERRMSG returns true if there was an I/O error (that is, if IORESULT isn't equal to ord(inoerror)). It also calls the system routine GETIOERRMSG (which is exported from module MISC) to put the proper English error message into a string parameter.

```
procedure iocheck;
begin if ioresult<>ord(inoerror) then escape(-10); end;
```

PROCEDURE IOCHECK tests to see if IORESULT is non-zero (<> ord(inoerror)) and if so,

escapes with escapecode = -10. IOCHECK should be called after any I/O operation which might fail, unless the caller wishes to explicitly test IORESULT to find out what happened.

```
procedure badio(iocode : iorsltd);
begin ioresult := ord(iocode); escape(-10); end;
```

PROCEDURE BADIO sets IORESULT to the specified value and escapes.

```
function unitnumber(var fvid : vid):boolean;
var scanning: boolean;
    i: shortint;
begin
    unitnumber := false;
    zapspace(fvid);
    if strlen(fvid) > 1 then
        if fvid[1]='#' then
            begin
                scanning := true; i := 2;
                repeat
                    if (fvid[i]>='0') and (fvid[i]<='9') then i := i + 1
                    else scanning := false;
                until (i>strlen(fvid)) or not scanning;
                unitnumber := scanning;
            end;
    end; { unitnumber }
```

FUNCTION UNITNUMBER tests a string to see if it is exactly a unit specification, that is, it has the form #ddd, where ddd is one or more digits. (The procedure ZAPSPACES removes blanks and control characters from the string.)

```
function samedevice(unit1,unit2:unitnum):boolean;
var
    u : ^unitentry;
begin
    u := addr(unitable^[unit1]);
    with unitable^[unit2] do
        samedevice := (u^.sc=sc) and (u^.ba=ba) and
            (u^.du=du) and (u^.dv=dv) and
            (u^.letter=letter) and
            (u^.byteoffset=byteoffset);
    end; { samedevice }
```

FUNCTION SAMEDEVICE compares some corresponding fields of two entries in the unit table to determine if they are the same physical device, such as an SRM or disk.

```
procedure anytomem(          ffib   : fibp;
                           anyvar buffer : bigptr;
                           maxbuf  : integer);
var
    bufrec   : ^string255;
    bufptr   : ^char;
    leftinbuf : integer;
```

```

begin { anytomem }
  bufptr := addr(buffer^);
  bufptr^ := chr(0); { data comming }
  bufrec := addr(bufptr^,1);
  setstrlen(bufrec^,0); { zero length record }
  bufptr := addr(bufrec^,1);
  leftinbuf := maxbuf;

  with ffib^, unitable^[funit] do
  begin
    call(am,ffib,readtoeol,bufrec^,255,fpos);
    repeat
      iocheck; { check ioresult from last readtoeol }
      bufptr := addr(bufptr^,strlen(bufrec^));
      leftinbuf := leftinbuf - strlen(bufrec^) - 2;
      if strlen(bufrec^) = 255 then bufptr := addr(bufptr^,-1)
      else
      begin
        if strlen(bufrec^)=0 then
        begin { discard the length byte }
          bufptr := addr(bufrec^,-1); leftinbuf := leftinbuf + 1;
        end;

        { check end of line/file }
        call(am,ffib,readbytes,bufptr^,1,fpos);
        if feoln then
        begin { end of line }
          bufptr^ := chr(1); feoln := false;
          if ioresult = ord(ieof) then bufptr := addr(bufptr^,1);
        end;
        if ioresult=ord(ieof) then
        begin { end of file }
          bufptr^ := chr(2);
          ioresult := ord(inoerror);
          feof := true;
        end;
        iocheck; { check ioresult from readbytes }
      end;
      if not ((leftinbuf < 259) or feof) then
      begin { setup for then read the next line }
        bufptr := addr(bufptr^,1);
        bufptr^ := chr(0); { data record }
        bufrec := addr(bufptr^,1);
        setstrlen(bufrec^,0); { zero length record }
        bufptr := addr(bufrec^,1);
        call(am,ffib,readtoeol,bufrec^,255,fpos);
      end;
      until (leftinbuf < 259) or feof;
      bufptr := addr(bufptr^,1); bufptr^ := chr(3); { end buffer }
    end;
  end; { anytomem }

```

PROCEDURE ANYTOMEM reads the source file into the buffer until it reaches the end of the file or until the buffer is full.


```

procedure memtoany(anyvar buffer : bigptr;
                  FFIB : fibp);
var
  bytes : integer;
  bufptr: ^char;

begin
  bufptr := addr(buffer^);
  with ffib^, unitable^[funit] do
  begin
    bytes := 0;
    repeat
      bufptr := addr(bufptr^,bytes);
      bytes := ord(bufptr^);
      bufptr := addr(bufptr^,1);
      case bytes of
        0: begin { data bytes }
            bytes := ord(bufptr^); { record length }
            bufptr:= addr(bufptr^,1);
            call(am,ffib,writebytes,bufptr^,bytes,fpos);
          end;
        1: begin { end record }
            call(am,ffib,wroteeol,bufptr^,bytes,fpos); bytes := 0;
          end;
        2: begin { end file }
            call(am,ffib,flush,bufptr^,bytes,fpos); bytes := -1;
          end;
        3: bytes := -1; { end buffer }
        otherwise ioreult := ord(ibadrequest);
      end;
      iocheck;
    until bytes<0;
  end;
end; { memtoany }

```

PROCEDURE MEMTOANY writes the contents of the buffer into the destination file.

PROCEDURES ANYTOMEM and MEMTOANY

These two procedures provide the mechanism by which files (primarily TEXT) are TRANSLATED. In the Pascal 2.0 system, text is not necessarily stored as a stream of ASCII characters; in fact, there are at least three distinct formats for text files. It is the job of routines called Access Methods (AM's) to construct the proper format for a text file when it is being written, and to interpret that format when it is being read back.

When a file is opened, the DAM (Directory Access Method) selects one of several possible AM's to use with it, according to the type of file. The entry point of that AM is stored in a procedure variable in the FIB (File Information Block). There is even a special AM for serial devices, such as the printer, keyboard, and screen.

All textual information in a Pascal file can be represented by strings of characters punctuated by end-of-line markers, and terminated by an end-of-file marker. Lines can be arbitrarily long (although some formats limit line length). The translation process consists of calling upon the AM for the source file to read strings of characters and to determine where the end-of-lines are,

then to call upon the AM for the destination file to write the characters and end-of-lines in the same sequence.

The characters are read into a large buffer, which consists of variable length records of information. A flag is placed at the beginning of each record to indicate what the record signifies, as follows:

0: A string of characters (bytes) to be written. The number of characters is given by the byte following the flag byte. The actual characters follow the length byte.

1: An end-of-line marker.

2: The end of the file.

3: The end of the buffer.

The AM service requests which are used by ANYTOMEM and MEMTOANY are as follows:

- **READTOEOL:** Reads a string of characters from the file into the buffer. Characters are read until an end-of-line is reached, or the end of the file, or until the maximum indicated number is reached (in this example, 255). The actual number of characters which are returned is indicated by a length byte preceding the characters themselves, so the format is that of a Pascal string.
- **READBYTES:** Reads the indicated number of characters (usually only 1) into the buffer. End-of-line markers are translated to spaces (ASCII 32). If the last character is an end-of-line, then the FEOLN flag in the FIB is set to true. If the end of the file is reached before the requested number of characters is read, then IORESULT is set to ord(IEOF).
- **WRITEBYTES:** Writes the indicated number of characters to the file. (No length byte is associated with these characters.)
- **WRITEEOL:** Writes an end-of-line marker into the file.
- **FLUSH:** Finishes writing to a file, done at the end of the file.

```
procedure setupfibforfile(var filename      : fid;
                          var lfib        : fib;
                          requiredirectory : boolean);
var
  lkind : filekind;
  segs  : integer;
begin
  ioresult := ord(inoerror);
  with lfib do
    if scantitle(filename,fvid,ftitle,segs,lkind) then
      begin
        funit      := findvolume(fvid,true);
        if funit = 0 then badio(inounit);
        if not ((ioresult = ord(inodirectory))
                and (strlen(ftitle) = 0)
                and (not requiredirectory))
        then begin
          iocheck;
```

```

        if unitnumber(fvid) then badio(znodevice);
        end;

        fkind      := lkind;      feft := ehtable^[lkind];
        fpos       := segs * 512;
        freptont   := 0;          flastpos := -1;
        fanonymous := false;      fmodified := false;
        fbufchanged := false;     fstartaddress := 0;
        pathid     := -1;         foptstring := nil;
        fnosrtemp  := true;       flocked := true;
        feof       := false;      feoln := false;
    end
    else badio(ibadtitle);
end;    { setupfibforfile }

```

PROCEDURE SETUPFIBFORFILE The major data structure associated with a file is the File Information Block, or FIB. Every open file must have its own FIB. The FIB contains all of the current state information associated with a file. The FIB is also the major communication path for requests to the DAM's (Directory Access Methods).

Initialization of the various fields of the FIB is similar for almost all file operations, so SETUPFIBFORFILE is called by many of the other routines in module FILEPACK.

The steps in initialization of the FIB are as follows:

1. Parse the file name into its component parts, i.e.
 - a. volume name (which might be a unit specifier)
 - b. file title (which may include the path name)
 - c. size specifier, if any
 - d. file type (derived from the suffix)
2. Determine what unit the file is associated with.
3. Assign known values to miscellaneous state variables.

The parsing is done by a system routine exported from module FS called SCANTITLE.

The FIB field FVID is reserved for the volume name, which is the part preceding the ':' in the filename (or the system prefix if it starts with '*', or the default prefix if no volume is specified).

The FIB field FTITLE contains the remainder of the file name except for the size specifier.

The size specifier (which appears in square brackets in the file name) is returned in the parameter SEGS; if it was absent, SEGS is 0, if it was '[*]', SEGS is -1.

The file kind is returned in the parameter LKIND. It is determined by looking up the suffix (e.g. '.TEXT') of the file title in a table.

Determining what unit the file name refers to is accomplished by a system routine exported from FS called FINDVOLUME. There are two cases: If the volume name is already a unit

specifier (e.g. '#12'), then the unit is obvious. In this case, FINDVOLUME calls the DAM for that unit to find out what is the actual current name of that unit (or the medium currently in the drive). It is usually an error if no name can be determined, but a special case is allowed if the only error is the absence of a directory when no file title was given and no directory is necessary (e.g. in the case of the CREATEDIR operation). This special case is allowed by setting the parameter REQUIREDIRECTORY to false.

The other case is when the volume name is not a unit specifier. In this case, FINDVOLUME searches all 50 units of the unit table to find a matching name. If it finds a match, it calls the DAM on that unit to verify that the name is still correct (the medium may have been removed or changed). If it does not find a match, it calls the DAM for all 50 units to verify the names of all units (just in case the required volume has recently come on line). If the volume cannot be found at all, FINDVOLUME returns 0 as the unit number. This is always an error. Otherwise, the unit number matching the volume name is returned and deposited in the FIB field FUNIT.

Initialization of the remaining FIB fields is simply a matter of giving them values which, by agreed upon conventions, correspond to the initial state of a file which is about to be opened. These fields are described elsewhere in the System Designer's Guide, but here are some brief comments about some of the fields:

FVID	the volume name
FTITLE	the file title
FUNIT	the logical unit number (indexes into the unit table)
FKIND	the file kind (e.g. text, data, code, etc.)
FEFT	the external file type, an integer code for the file type which is recognized by LIF and other HP file systems
FPOS	initially contains the size specifier from the file name, but later corresponds roughly to the file position. (This is always in bytes, so the size specifier must be multiplied by 512.)
FANONYMOUS	indicates a temporary file with no name (e.g. REWRITE(F))
FSTARTADDRESS	the execution address for type SYSTM (bootable) files
FOPSTRING	a pointer to the optional third parameter in open statements, a string which may contain, e.g., 'SHARED' or 'EXCLUSIVE'. may be initialized to NIL if no third parameter is given.
FMODIFIED	indicates that some attribute of the file has changed which would require modifying the directory (e.g. FLEOF, file size)
PATHID	often used by hierarchical directories (e.g. SRM) to store an identification code for the parent directory (the path name part of the file name). Must be initialized to -1.
FLOCKED	indicates whether a lockable file is currently locked
FEOF	indicates that the file position is at the end of the file
FEOLN	indicates that an end-of-line marker has been read
FREPTCNT	temp field used by AM's, must initially be 0
FLASTPOS	temp field used by AM's, must initially be -1
FBUFCHANGED	temp field used by AM's, must initially be false
FNOSRMTEMP	temp field used by SRM, must initially be true

```

procedure closeinfile(var infib: fib);
begin
  with infib do if freadable then
    begin
      fmodified := false;
      call(unitable^[funit].dam,infib,funit,closefile);
      freadable := false;
    end;
end;    { closeinfile }

```

PROCEDURE CLOSEINFILE calls the DAM to close a file which has been open for reading (indicated by the FREADABLE flag). FMODIFIED is set to false to insure that the file isn't altered.

```

procedure closeoutfile(var outfib: fib; option : closecode);
var
  coption : damrequesttype;
begin
  with outfib do if fwriteable then
    begin
      case option of
        keepit: begin
          fmodified := true;
          coption := closefile;
        end;
        purgeit: coption := purgefile;
      end;

      call(unitable^[funit].dam,outfib,funit,coption);
      fwriteable := false;
    end;
end;    { closeoutfile }

```

PROCEDURE CLOSEOUTFILE calls the DAM to close a file which has been open for writing (indicated by the FWRITEABLE flag). There are two options:

1. We wish to retain the file and make it permanent. This is usually when we have successfully completed a copy or translate. FMODIFIED is set to true to indicate a possible directory update. The DAM request is CLOSEFILE, which will make the file permanent in the directory, and purge any existing file of the same name.
2. We wish to delete this file. This is usually when we wish to abort an unsuccessful operation due to an error. The DAM request is PURGEFILE, which will delete this file, which is usually a temporary file. Any existing permanent file of the same name is **not** purged.

```

procedure filecopy(filename1, filename2: fid; format, writeover: boolean);
type
  fullname = string[vidleng+tidleng+1];
  ipointer = ^integer;
var
  infib, outfib : fib;
  outsize       : integer;
  outfkind      : filekind;

```

```

outeft      : shortint;
outfstarta  : integer;

overcreate  : damrequesttype;
typecode    : integer;

lheap       : anyptr;
saveio      : integer;
saveesc     : integer;

buf         : bigptr;
bufsize     : integer;
movesize    : integer;

begin { filecopy }
  mark(lheap);

  if format then typecode := -3 { TEXT file }
    else typecode := 1; { DATA file }

  newwords( infib.fwindow,1); { buffer variable }
  finitb( infib, infib.fwindow, typecode);

  newwords(outfib.fwindow,1); { buffer variable }
  finitb(outfib,outfib.fwindow, typecode);

  try
  with infib do
    begin
      setupfibforfile(filename1, infib, false);

      if strlen(ftitle)=0 then
        begin { volume -> x }
          call(unitable^[funit].dam,infib,funit,openvolume);
          fkind      := datafile;      feft := efttable^[datafile];
        end
      else begin { file -> x }
          call(unitable^[funit].dam,infib,funit,openfile);
        end;
      iocheck;

      fpos := 0;
      freadable := true;

      outfkind := fkind;
      outeft := feft;
      outsize := fleof;
      outfstarta := fstartaddress;
    end; { with infib }

  with outfib do
    begin
      setupfibforfile(filename2, outfib, false);

      if format then
        begin

```

```

    fkind := suffix(ftitle); { set destination fkind }
    feft := ehtable^[fkind];
    outside := 0;
    end
else
    begin
        fkind      := outfkind;
        feft      := outeft;
        if fpos = 0                {no size was specified}
            then fpos      := outside;
        end;
    fstartaddress := outfstarta;

    if strlen(ftitle)=0 then
        begin { x -> volume }
            call(uitable^[funit].dam,outfib,funit,openvolume);
            iocheck;
            if fpeof<outside then badio(inoroom);
            end
        else
            begin { x -> file }
                if writeover then overcreate := overwritefile
                    else overcreate := createfile;
                call(uitable^[funit].dam,outfib,funit,overcreate);
                iocheck;
                if fpeof<outside then
                    begin { try to stretch the file }
                        fpos := outside;
                        call(uitable^[funit].dam,outfib,funit,stretchit);
                        iocheck;
                        if outside>fpeof then badio(inoroom);
                        end;
                    end;
            end;

            fpos := 0;
            fwriteable := true;
        end;

    bufsize := ((memavail- 5000) div 256) * 256; {save 5K for slop}
    if bufsize<512 then escape(-2); { not enough room }
    newwords(buf,bufsize div 2); { allocate buffer space }

    if format then outside := -1;

    repeat { move the file }
        with infib do
            if format then
                begin { formatted filecopy }
                    anytomem(addr(infib),buf,bufsize);
                    if feof then outside := 0;
                    end
                else
                    begin { unformatted filecopy }
                        if bufsize>outside then movesize := outside
                            else movesize := bufsize;
                        call(uitable^[funit].tm,addr(infib),readbytes,buf^,movesize,fpos);
                    end
                end
    end

```

```

        fpos := fpos + movesize;
    end;
iocheck;

with outfib do
    if format then
        memtoany(buf,addr(outfib))
    else
        begin { unformatted filecopy }
            call(unitable^[funit].tm,addr(outfib),writebytes,
                buf^,movesize,fpos);
            fpos := fpos + movesize;
            fleof := fpos;
            outside := outside - movesize;
        end;
    iocheck;

until outside = 0;

release(lheap);
closeinfile(infib);
closeoutfile(outfib, keepit);

recover
begin
    release(lheap);
    saveio      := ioresult;
    saveesc     := escapecode;
    closeinfile(infib);
    closeoutfile(outfib, purgeit);
    ioresult    := saveio;
    escape(saveesc);
end;
end; { filecopy }

```

PROCEDURE FILECOPY provides the ability to copy or translate one file to another. **FILECOPY** will also copy whole volumes. **Filename1** is the name of the source file (or volume) and **filename2** is the name of the destination file (or volume). The boolean parameter **FORMAT** indicates that is a translate rather than a copy. The boolean parameter **WRITEOVER** indicates that the destination file should be overwritten.

The major steps are as follows:

1. Initialize the source and destination FIB's by calling the system procedure **FINITB** (which is exported from **FS**). This procedure initializes some FIB fields (e.g. **FISTEXTVAR** and **FISBUFFERED**) which are necessary for the DAM to properly choose the correct Access Method (See **ANYTOMEM**, etc.) for text files. The call to **newwords** is to get a dummy buffer variable which **FINITB** uses to initialize **FWINDOW**, although this won't be used by **FILECOPY**.
2. Open the source file. Notice that the call to **SETUPFIBFORFILE** does not require a directory to be present on the volume. There are two cases: If there is no file title part of the file name, then we assume that the source is an entire volume. The DAM is called using the **OPENVOLUME** request, which treats the whole volume as one big file. In this way, a whole disk can be copied. There doesn't even have to be a directory on it. Notice that the

file type is assumed to be DATA. If there is a file title present, call the DAM using the OPENFILE request. This searches the volume for an existing file of that name.

3. Set FPOS to 0 to indicate that reading will start at the beginning of the file. Also set FREADABLE to true to indicate that the file is open.
4. Extract the file type, size, and start address information from the FIB. These values may be necessary to use when opening the destination file.
5. Set up the FIB for the destination file. Again, no directory is required for a volume transfer. If this is a translate, then the destination file type is taken from the destination file name according to its suffix. In this case the size of the destination file cannot be determined from the size of the source file. If this is not a translate, then the file type information is adopted from the source file. If no size specifier was given in the destination name (FPOS = 0) then the file size is also taken from the source size. In both cases, FSTARTADDRESS is copied from the source file.
6. Open the destination file. If this is a volume transfer, the DAM request OPENVOLUME is used. A quick test of FPEOF (physical size of file) indicates whether the destination volume is big enough. If this is a file transfer, then according to the WRITEOVER parameter the file is either overwritten using the OVERCREATE request, or a new file is created using the CREATEFILE request. If the file just opened isn't big enough, an attempt is made to increase its size by calling the DAM using the STRETCHIT request. FPOS indicates the desired size for stretching. If stretching fails there is no more that can be done.
7. Set FPOS to 0 to indicate that writing will start at the beginning of the file. Set FWRITEABLE to true to indicate that the file is open.
8. Allocate a large buffer. Grab the memory in increments of 256 bytes, since this is the size of physical sectors on most HP mass storage media. Take as much memory as is available, except leave enough for the program's execution stack space (5K should be enough). Set ousize to -1 for translating because it is used as a flag to terminate the transfer loop.
9. Read as much of the source file as will fit into the buffer. If translating, use PROCEDURE ANYTOMEM. FEOF indicates the end of the file has been reached. If copying, call the device driver directly. The device driver is called a Transfer Method (TM) and its address is stored in the unit table. FPOS indicates the current file position and it must be advanced by the number of bytes read.
10. Write the contents of the buffer to the destination file. If translating use PROCEDURE MEMTOANY, otherwise call the device driver (TM). FPOS must be advanced to indicate the file position. Set FLEOF to indicate the logical file size.

Repeat steps 9) and 10) until the whole file is transferred.

11. Release the buffer memory.
12. Close the input file.
13. Close the output file with the option to make it permanent.

```
procedure volumes(var v: volumearray);  
var  
  un      : unitnum;
```

```

i      : integer;
sym    : string[3];

begin
  for un := 1 to maxunit do
  with unitable^[un] do
    begin
      call(dam, uvid, un, getvolumename);
      v[un] := '';
      if (ioresult=ord(inoerror)) and (strlen(uvid) > 0) then
        begin
          if uvid = syvid then sym := ' * '
          else
            if uisblkd then sym := ' # ' else sym := '   ';
            strwrite(v[un], 1, i, sym, uvid, ':');
          end;
        end;
      end;
    end;
  { volumes }
end;

```

PROCEDURE VOLUMES creates a list of the volumes which are on line. The list is generated into an array of 50 strings. The index of the string corresponds to the unit number. A null string indicates a unit which is not on line.

The procedure is simply to loop through all the units and for each one call the DAM using the GETVOLUMENAME request. Notice that no FIB is needed for this request, and the volume name is returned as a string parameter in the position normally occupied by the FIB parameter. The string UVID is a field in the unit table which normally records the volume name. The system volume is recognized because its volume name is the same as the global variable SYVID. Blocked volumes are recognized by the field in the unit table named UISBLKD.

```

procedure repack(filename: fid);
var infib: fib;
begin
  with infib do
    begin
      setupfibforfile(filename, infib, true);
      call(unitable^[funit].dam, infib, funit, crunch);   iocheck;
    end;
  end;
  { repack }
end;

```

PROCEDURE REPACK calls on the DAM using the CRUNCH request to repack the indicated volume.

```

procedure opendir(var filename      : fid;
                 var infib         : fib;
                 var dircatentry    : catentry);

begin { opendir }
  with infib do
    begin
      freadable := false;
      fwindow   := addr(dircatentry);
      setupfibforfile(filename, infib, false);
      if ioresult = ord(inoerror) then

```

```

begin
  call(unitable^[funit].dam,infib,funit,opendirectory);
  ioccheck;
  freadable := true;
end;
end;
end; { opendir }

```

PROCEDURE OPENDIR: The major data structure for creating and cataloging directories is the **CATENTRY**. A **CATENTRY** contains fields which give information about the directory such as its name, the number of files it can handle, etc. (See **PROCEDURE STARTCAT**.) **PROCEDURE OPENDIR** calls the **DAM** to do an operation called **OPENDIRECTORY**, which is similar to opening a file except that it prepares the directory for operations such as cataloging. The call to the **DAM** does several things:

1. The field in the **FIB** called **FWINDOW** must be a pointer to a **CATENTRY**. The **DAM** fills in the fields of the **CATENTRY** with information about the directory.
2. The file title in the field **FTITLE** is parsed to determine whether part or all of it is the name of a file (as opposed to the path name for a hierarchical directory). Only the file name part is returned in **FTITLE**.

For example, if **FTITLE** was **'/USERS/JOHN/TEST1.TEXT'**, then the **OPENDIRECTORY** would return **'TEST1.TEXT'** in **FTITLE**, and it would place information about the directory called **'JOHN'** into the **CATENTRY** pointed to by **FWINDOW**, as well as preparing **'JOHN'** for cataloging.

```

procedure closedir(var infib : fib);
begin
  with infib do
  begin
    if freadable then
    begin
      call(unitable^[funit].dam,infib,funit,closedirectory);
      freadable := false;
    end;
  end;
end; { closedir }

```

PROCEDURE CLOSEDIR calls the **DAM** to perform a **CLOSEDIRECTORY** operation, which is similar to closing a file except that it refers to a directory which was opened by an **OPENDIRECTORY**. Other operations, such as cataloging, may be done before closing the directory, but a **CLOSEDIRECTORY** must always be done eventually or else the file system may hold that directory open (which might block other operations or other users from accessing that directory).

```

procedure createdir(filename: fid; newname: vid; entries, bytes: integer);
var
  infib          : fib;
  dircatentry    : catentry;
  saveio, saveesc: integer;

begin { createdir }
  with infib, dircatentry do

```

```

try
opendir(filename, infib, dircatentry);
if ioresult = ord(inodirectory) then
  begin
    { no directory, so setup default values}
    setstrlen(cname,0);      {volume name}
    cpsize := maxint;      {size in bytes}
    cextral := 0;          {number of entries}
  end
else if (strlen(ftitle)>0) or (cpsize<=0) then badio(ibadrequest);

closedir(infib);

cpsize := min(cpsize, ueovbytes(funit));

if entries >= 0 then cextral := entries;      { -1 retains old value}
                                           { 0 selects default}

if bytes > 0 then cpsize := bytes;          { -1 retains old value}
if cpsize=0 then badio(ibadvalue);

zapspace(newname);
if strlen(newname) > 0 then cname := newname; { null retains old name}

call(unitable^[funit].dam,infib,funit,makedirectory);
iocheck;
recover begin
  saveio      := ioresult;
  saveesc     := escapecode;
  closedir(infib);
  ioresult    := saveio;
  escape(saveesc);
end;

end; { createdir }

```

PROCEDURE CREATEDIR creates a new directory on a mass storage medium. The parameters needed are:

1. The filename of the device to be zeroed (usually a unit number)
2. The new name of the volume to be created
3. The maximum number of directory entries which are needed
4. The maximum size in bytes of the medium or logical volume

CREATEDIR first calls **PROCEDURE OPENDIR** to find out if there is already a directory on the volume. If there is, information about it is placed in the **CATENTRY**:

- **CNAME** is the volume name.
- **CEXTRAL** is the number of possible directory entries.
- **CPSIZE** is the physical size of the volume in bytes. If **CPSIZE** is 0, it indicates that it is a type of device for which zeroing a volume directory is inappropriate, so this is an error.

- If FTITLE contains a non-null string, it indicates a file name was present in the original file name. This is an error. At this point, PROCEDURE CLOSEDIR is called to close the directory, since all the useful information has been extracted. The system function UEOVBYES (exported from module MISC) is called to check the maximum size in bytes of the volume. If there was a previous directory, then the same name as the old one can be retained by passing a null string for the new name. The same number of directory entries can be retained by passing -1. A default number of directory entries (dependent on the particular DAM) can be selected by passing 0. The same physical volume size can be retained by passing -1.
- Finally, the DAM is called using the MAKEDIRECTORY request.

```

procedure makedir(filename: fid);
var
  infib          : fib;
  dircatentry   : catentry;
  saveio, saveesc: integer;

begin
  with infib, dircatentry do
    try
      opendir(filename, infib, dircatentry);
      iocheck;
      if strlen(ftitle)=0 then badio(idupfile);
      cname := ftitle;
      call(unitable^[funit].dam, infib, funit, makedirectory);
      iocheck;
      closedir(infib);
    recover begin
      saveio      := ioresult;
      saveesc     := escapecode;
      closedir(infib);
      ioresult    := saveio;
      escape(saveesc);
    end;
  closedir(infib);
end;

```

PROCEDURE MAKEDIR makes a directory, usually on a hierarchically structured device such as an SRM. The only parameter needed is the new file name. First, OPENDIR is called to test whether the given directory already exists, which would be indicated by having no file name returned in FTITLE. The string returned in FTITLE will be the name of the new directory, so it must be placed in the CNAME field of the CATENTRY. Then the DAM is called using the MAKEDIRECTORY request.

```

procedure makefile(filename: fid);
var
  outfib        : fib;

begin
  with outfib do
    begin
      setupfibforfile(filename, outfib, true);
      call(unitable^[funit].dam, outfib, funit, createfile);
    end;
end;

```

```

        iocheck;
        fwriteable := true;
        fleof := fpeof;           {cause file size to be retained}
        closeoutfile(outfib,keepit);
        iocheck;
    end; { with }
end; { make }

```

PROCEDURE MAKEFILE creates an empty file by calling the DAM using the request CREATEFILE, which also opens it. The flag FWRITEABLE is set to true to indicate that the file has been opened. The FIB field FLEOF records the Logical End Of File (the file size) in bytes. This is always zero for a newly created file, so it must artificially be set to the Physical End Of File (FPEOF, the amount of disk space allocated to the file) even though no contents have been written to the file. This forces the file to be of the size given in the size specifier in the file name, if there was one. PROCEDURE CLOSEOUTFILE is called to make the file permanent.

```

procedure endcat;
begin
    if catfib <> nil then
        begin
            closedir(catfib^);
            release(catfib);
            catfib := nil;
        end;
    end;
end;

```

PROCEDURE ENDCAT deallocates the heap space that was allocated by PROCEDURE STARTCAT in order to do cataloging. It also calls PROCEDURE CLOSEDIR to release the directory being cataloged. Naturally, this procedure should always be called when you have finished cataloging.

```

procedure startcat(filename: fid;
    var dirname: vid;
    var typeinfo: string;
    var createdate, changedate: daterec;
    var createtime, changetime: timerec;
    var blocksize, phy_size, start_byte, free_bytes, max_files: integer);

var
    dircatentry    : catentry;
    saveio         : integer;
    saveesc        : integer;

begin { listdir }
    endcat;
    new(catfib);
    new(catentptr);
    try
        opendir(filename,catfib^,dircatentry);
        iocheck;
        with dircatentry do
            begin
                dirname := cname;

```

```

    typeinfo := cinfo;
    createdate := ccreatedate;   changedate := clastdate;
    createtime := ccreatetime;   changetime := clasttime;
    blocksize := cblocksize;
    phy_size   := cpsize;
    start_byte := cstart;
    free_bytes := cextra2;
    max_files  := cextra1;
    end;
with catfib^, unitable^[funit] do
    begin
    fwindow := addr(catentptr^);
    fpos := 0;
    fpeof := catlimit;
    call(dam, catfib^, funit, catalog);
    iocheck;
    end;
recover
    begin
    saveio      := ioresult;
    saveesc := escapecode;
    endcat;
    ioresult := saveio;
    escape(saveesc);
    end;
end;

```

PROCEDURE STARTCAT initiates a cataloging operation. The only input parameter is the name of the directory to be cataloged. The following parameters return useful information about the directory:

DIRNAME	the name of the directory or volume
TYPEINFO	a string up to 20 characters, usually labelling the kind of directory it is, e.g. LIF, SRM etc.
CREATEDATE	the date that the directory was created
CREATETIME	the time that the directory was created
CHANGEDATE	the date that the directory was last modified
CHANGETIME	the time that the directory was last modified
BLOCKSIZE	the size in bytes of a logical block, usually 256, 512, or 1
PHY_SIZE	the physical size of the volume
START_BYTE	the first possible byte offset of a file on the volume
FREE_BYTES	the amount of remaining space available for files
MAX_FILES	the maximum number of files the directory can hold, if limited

Not all of the items above may be applicable to every directory type, so some of the values returned may be -1 or 0 to indicate they don't apply.

All of the above items are returned in fields of the CATENTRY record which is pointed to by FWINDOW. (See the code for the specific field names.)

The steps of initiating a catalog are as follows:

1. Allocate the FIB from the heap.

2. Allocate CATENTPTR, a pointer to an array of CATENTRY records.
3. Initialize the FIB by calling PROCEDURE OPENDIR. (FWINDOW becomes a pointer to a local CATENTRY called DIRCATENTRY)
4. Extract the information about the directory from the fields of the CATENTRY into the return parameters.
5. Initialize these fields of the FIB: FWINDOW should be changed to point to the array, CATENTPTR^ FPOS indicates which file should be cataloged first (initially 0) FPEOF indicates how much room there is for cataloging files
6. Call the DAM using the CATALOG request
7. Upon successful completion, the array of CATENTRY records has been filled with information about the files in the directory, up to the maximum limit given. FPEOF returns with the actual number of files which were cataloged.

```

procedure cat(filename: integer;
  var fileinfo: string;
  var createdate, changedate: daterec;
  var createtime, changetime: timerec;
  var kind: filekind;
  var eft: shortint;
  var blocksize, logical_size,
    phy_size, start_byte,
    extension1, extension2: integer);
begin
  if catfib = nil then escape(-3);
  with catfib^, unitable^[funit] do
    begin
      if not freadable then badio(inotopen);
      if (filename >= 0) and
        ((filename < fpos) or ((filename >= fpos + catlimit) and (fpeof = catlimit)))
      then begin
        fpos := filename; fpeof := catlimit;
        call(dam, catfib^, funit, catalog);
        iocheck;
        end;
      if (filename < fpos) or (filename >= fpos + fpeof) then filename := ''
      else with catentptr^[filename - fpos] do
        begin
          filename := cname;
          typeinfo := cinfo;
          createdate := ccreatedate; changedate := clastdate;
          createtime := ccreatetime; changetime := clasttime;
          kind := ckind;
          eft := ceft;
          blocksize := cblocksize; logical_size := clsize;
          phy_size := cpsize; start_byte := cstart;
          extension1 := cextra1; extension2 := cextra2;
        end;
      end;
    end;
end;

```


PROCEDURE CAT retrieves the catalog information about a particular file. You must call this routine only after having initiated a catalog using PROCEDURE STARTCAT (but you may call CAT as many times as you like without calling STARTCAT again). The input parameter to CAT is the parameter FILENUMBER, an integer from 0 to N-1, if there are N files in the directory. Information about the file is returned in the following parameters:

FILENAME	the name of the file. If this is a null string, it indicates that there are no more files.
TYPEINFO	a string of up to 20 characters, giving miscellaneous information about the file, such as protect codes or whether the file is open
CREATEDATE	the date that the file was created
CREATETIME	the time that the file was created
CHANGEDATE	the date that the file was last modified
CHANGETIME	the time that the file was last modified
KIND	the file kind, e.g. data, code, text, etc.
EFT	the external file type, an HP standard type code
BLOCKSIZE	the size of a logical block, usually, 256, 512, or 1
LOGICAL_SIZE	the size of the file in bytes
PHY_SIZE	the physical size of the file, i.e. the allocated disk space
START_BYTE	the byte offset of the start of the file from the beginning
EXTENSION1	implementation dependent information, depends on directory type
EXTENSION2	implementation dependent information, depends on directory type

Not all of the items above may be applicable to every directory type, so some of the values returned may be -1 or 0 to indicate they don't apply.

The information about the file is in the element of the array of CATENTRY (see the code for the specific field names) which is indexed by (FILENUMBER-FPOS) as long as FPOS <= FILENUMBER <= (FPOS+FPEOF). If FILENUMBER is not in this range, then another call to the DAM using the CATALOG request must be made (but notice that there is no need to try it if FPOS is too large and FPEOF is not equal to the maximum value CATLIMIT, since this means there aren't any more files).

```

procedure duplicate(filename1, filename2: fid; purgeold: boolean);
var
  infib, outfib: fib;
  dircatentry  : catentry;
  saveio, saveesc: integer;

begin
  with infib do
    try
      opendir(filename1, infib, dircatentry);
      iocheck;
      opendir(filename2, outfib, dircatentry);
      iocheck;
      if not samedevice(funit, outfib.funit) then badio(ibadrequest);
      fwindow := addr(outfib);
      fpurgeoldlink := purgeold;
      call(unitable^[funit].dam, infib, funit, duplicatelink);
      iocheck;
      closedir(infib);
    
```

```

closedir(outfib);
recover
begin
  saveio      := ioreult;
  saveesc     := escapecode;
  closedir(infib);
  closedir(outfib);
  ioreult     := saveio;
  if saveesc <> 0 then escape(saveesc);
end;
end; { duplicate }

```

PROCEDURE DUPLICATE makes a duplicate link to a file, on those directory types which support links to files (e.g. SRM). The parameter PURGEOLD gives the option of purging the link to the source file, so that the effect is that of moving the file from one directory to another. Of course, the source and destination must both be on the same device, since this is an operation which merely manipulates links in the (hierarchical) directory.

For this operation two FIB's are used. The DAM is called to do an OPENDIRECTORY for both the source and destination directories. The field FWINDOW of the source FIB is a pointer to the destination FIB. There is a field in the source FIB called FPURGEOLDLINK which indicates that this is a move rather than a duplicate.

```

procedure remove(filename: fid);
var infib: fib;
begin
  setupfibforfile(filename, infib, true);
  with infib do
    call(unitable^[funit].dam, infib, funit, purgename);
  iocheck;
end;

```

PROCEDURE REMOVE purges a file by calling the DAM with the PURGENAME command. The only advantage to doing it this way instead of with the standard Pascal sequence: RESET(F, 'FILENAME'); CLOSE(F, 'PURGE'); is that it is only one call to the DAM and doesn't actually open the file.

```

procedure change(filename1, filename2: fid);
var infib, outfib: fib;
    lsegs: integer;
    lkind: filekind;
begin
  setupfibforfile(filename1, infib, true);
  with outfib do
    if not scantitle(filename2, fvid, ftitle, lsegs, lkind) then badio(ibadttitle);
  with infib do
    if ftitle = '' then
      call(unitable^[funit].dam, outfib.fvid, funit, setvolumename)
    else begin
      fwindow := addr(outfib.ftitle);
      call(unitable^[funit].dam, infib, funit, changename);
    end;
  iocheck;
end; { change }

```

PROCEDURE CHANGE changes the name of a file or a volume. The system procedure SCANTITLE (exported from FS) is used to parse the new name into its component parts. There are two cases:

1. If there is no file title part in the original name, it is assumed we are changing a volume name. In this case, the DAM is called using the SETVOLUMENAME request. The new volume name is passed to the DAM in the parameter position normally occupied by the FIB; the FIB is not actually used for this request.
2. If there is a file title part in the original name, then we are changing a file name. In this case, the field FWINDOW of the FIB is made to be a pointer to the (title part of) the new file name. Then the DAM is called using the CHANGENAME request.

```
procedure endpass;
begin
  if passfib <> nil then release(passfib);
end;
```

PROCEDURE ENDPASS releases the heap memory which was allocated by PROCEDURE STARTLISTPASS in order to manipulate passwords. ENDPASS should always be called after listing and changing passwords is complete.

```
procedure startlistpass(filename: fid);
begin
  endpass;
  new(passfib);
  new(wordlist);

  try
    setupfibforfile(filename, passfib^, true);
    with passfib^ do
      begin
        fwindow := addr(wordlist^);
        fpos := 0; fpeof := catlimit;
        call(unitable^[funit].dam, passfib^, funit, catpasswords);
        iocheck;
        optionlist := addr(foptstring^);
      end;
    recover
      begin
        endpass; escape(escapecode);
      end;
  end;
```

PROCEDURE STARTLISTPASS initiates the process of listing the current passwords of a file. The steps are:

1. Allocate the FIB from the heap.
2. Allocate WORDLIST, a pointer to an array of PASSETRY records.

3. Initialize the FIB:
 - a. FWINDOW is a pointer to WORDLIST^
 - b. FPOS indicates which password should be listed first (initially 0)
 - c. FPEOF indicates how much room there is for listing passwords
4. Call the DAM using the CATPASSWORDS request
5. Upon successful completion, the FIB field FOPTSTRING is a pointer to an array of PASSETRY records which enumerate the allowed, legal attributes for this particular directory type. The value in the PBITS field is a bit pattern which is used to associate these attributes with passwords. A PBITS field of 0 marks the end of the list.
6. The array WORDLIST has been filled with the current passwords of the file (up to the limit given in FPEOF). FPEOF gives the actual number of passwords returned. The PBITS field of the PASSETRY is a bit pattern which is used to match these passwords with their attributes.

```

procedure listattribute(wordnumber: integer; var outstring: string);
var i: integer;
    done: boolean;
begin
  outstring := '';
  if passfib = nil then escape(-3);
  with passfib^ do
  begin
    i := 0;
    done := false;
    repeat
      with optionlist^[i] do
      begin
        if pbits = 0 then done := true
        else if i = wordnumber then begin
          outstring := pword;
          done := true;
          end;
        end;
        i := i + 1;
      until done;
    end;
  end;
end;

```

PROCEDURE LISTATTRIBUTE searches the list of legal password attributes produced by PROCEDURE STARTLISTPASS and returns the name of the one which is the Nth attribute, into the string parameter OUTSTRING. WORDNUMBER is an integer from 0 to N-1 which indicates which attribute you want. OUTSTRING is set to null if N is too large.

```

procedure listpassword(wordnumber: integer; var outstring: string);
var i, j, p: integer;
    first, last: boolean;
begin

```

```

outstring := '';
if passfib = nil then escape(-3);
with passfib^ do
begin
if (wordnumber >= 0) and
((fwindow <> addr(wordlist^)) or
(wordnumber < fpos) or ((wordnumber>=fpos+catlimit) and (fpeof=catlimit)))
then begin
fwindow := addr(wordlist^);
fpos := wordnumber; fpeof := catlimit;
call(unitable^[funit].dam, passfib^, funit, catpasswords);
iocheck;
end;
if (wordnumber >= fpos) and (wordnumber < fpos + fpeof) then
with wordlist^[wordnumber-fpos] do
if pbits <> 0 then
begin
strwrite(outstring, 1, j, pword, ',');
first := true;
last := false;
i := 0;
p := pbits;
repeat
with optionlist^[i] do
begin
last := pbits = 0;
if not last then if iand(pbits, p) = pbits then
begin
if not first then strwrite(outstring, strlen(outstring)+1, j, ',');
first := false;
strwrite(outstring, strlen(outstring)+1, j, pword);
end;
end;
i := i + 1;
until last;
end;
end;
end;
end;
end;
end;

```

PROCEDURE LISTPASSWORD returns information about the Nth password of the file whose passwords are being listed. The desired information is in the PASSEnTRY record indexed by (WORDNUMBER-FPOS) as long as FPOS <= WORDNUMBER < (FPOS+FPEOF). If wordnumber is out of this range, or if FWINDOW no longer points to the WORDLIST array due to having modified a password, then a new call to the DAM to do a CATPASSWORDS must be made. If WORDNUMBER is still out of range, the null string is returned; otherwise the parameter OUTSTRING will be constructed in the form:

PASSWORD:ATTRIBUTE,ATTRIBUTE,ATTRIBUTE etc.

The attributes are matched to the password by scanning the list of legal attributes. An attribute is associated with the password if the PBITS field of the attribute is a logical subset of the PBITS field of the password.

```

procedure changepassword(word: passtype; attrlist: string255);
var entry: passentry;

```

```

    name: passtype;
    bits,i: integer;
    found: boolean;
begin
    if passfib = nil then escape(-3);
    bits := 0;
    zapspace(attrlist); {remove blanks and control characters}
    while strlen(attrlist) > 0 do
        begin
            i := strpos(',',attrlist);
            if i=0 then i := strlen(attrlist) + 1;
            name := str(attrlist,i,i - 1); upc(name); { uppercase the attribute }
            if i > strlen(attrlist) then setstrlen(attrlist, 0)
            else attrlist := str(attrlist,i+1,strlen(attrlist)-i);
            i := 0;
            found := false;
            repeat
                with optionlist^[i] do
                    begin
                        if pbits = 0 then badio(ibadformat);
                        if name = pword then begin
                            found := true;
                            bits := ior(bits, pbits);
                            end;
                        end;
                    i := i + 1;
                until found;
            end;          { get attributes }

        zapspace(word);
        with entry do
            begin
                pword := word;
                pbits := bits;
            end;
        with passfib^ do
            begin
                fwindow := addr(entry);
                fpos := 0; fpeof := 1;
                call(unitable^[funit].dam, passfib^, funit, setpasswords);
                iocheck;
            end;
        end;
end;

```

PROCEDURE CHANGEPASSWORD allows passwords to be added, deleted, or associated with a different set of attributes. The password to be modified is given in the parameter WORD. The new set of attributes is given by the string parameter ATTRLIST as a list separated by commas. ATTRLIST should be a null string to specify that the password is to be deleted. The steps are as follows:

1. Construct a PBITS bit map by logically ORing together the PBITS fields of each of the legal attributes which appear in the ATTRLIST string.
2. Put the password WORD together with its new PBITS value into a PASSEENTRY record

3. Set these fields in the FIB:

- a. FWINDOW is a pointer to the new PASSEENTRY record
- b. FPOS is 0
- c. FPEOF is 1

4. Call the DAM using the SETPASSWORDS request

```
procedure prefix(filename: fid; unitonly, sysvol: boolean);
var i: integer;
    s: vid;
begin
  zapspaces(filename);
  if unitonly then doprefix(filename, s, i, true)
  else if sysvol then doprefix(filename, syvid, sysunit, true)
  else doprefix(filename, dkvid, i, false);
  ioccheck;
end;
```

PROCEDURE PREFIX uses a system routine exported from FS called DOPREFIX to change either the default prefix, or the system volume, or the prefix on an individual unit, according to these parameters:

	unitonly:	sysvol:
unit:	TRUE	(don't care)
system:	FALSE	TRUE
default:	FALSE	FALSE

The variables SYVID, SYSUNIT, and DKVID are global variables exported from SYSGLOBALS. SYVID is the name of the system volume, SYSUNIT is the unit number of the system volume, and DKVID is the name of the default volume. DOPREFIX returns values into these variables.

```
end. { module filepack }
```

Writing your Own Command Interpreter

The "Command Interpreter" is the program which displays the outer-level menu on the CRT, allowing the user to invoke subsystems such as the Editor and Compiler. This program is found on the boot device under the name STARTUP. It is loaded and executed by TAIL, the last module in INITLIB.

STARTUP may be any application program -- there need not be anything special about it. For instance, if you write an accounting program named STARTUP and put it on the BOOT: disc, the "system" will come up running that accounting program. If the program terminates for any reason, the message "SYSTEM FINISHED. RESET TO RESTART" will be displayed. At that point the computer must be re-booted.

The Standard Command Interpreter

The standard Command Interpreter provided with Pascal 2.0 is a program called CMD, which invokes code provided in module CI. These together make up the standard STARTUP. The code is not very interesting, and in fact is not included in this documentation. Instead, this section shows you how to write your own CI.

However, there are three routines exported from the standard CI which may be useful. They are:

procedure chain (filename: fid);

This routine accepts a file specifier which should be the name of a program. If the currently executing program terminates normally after calling chain, the standard CI will then run the named program.

procedure startstream (filename: fid);

This routine accepts a file specifier which should be the name of a text file, eg 'DO_ME.TEXT'. After the call to STARTSTREAM, any input characters which would normally have been read from the standard file INPUT are instead taken from the named file, until end-of-file is reached. When EOF is reached, input reverts to the keyboard.

If the system was already streaming when STARTSTREAM is called, the current stream input file is closed and the newly specified one is opened. Stream files cannot be "nested".

function streaming: boolean;

This function returns true if and only if input is currently being taken from a stream file.

Creating a New Command Interpreter

On the other hand, you may be writing an out-of-the-ordinary application, which needs to have features more like those of the standard command interpreter. In particular, a user "shell" frequently needs to load or execute other programs. The purpose of this chapter is to show how to do that. By combining the information in this chapter with the capabilities provided by Directory Access Method calls, it is possible to write very powerful, interesting and "user-friendly" applications.

Because so many of the necessary capabilities are exported from various system modules, a minimal command interpreter capable of executing programs and cleaning up after errors can be written in about 100 lines of Pascal. The "custom" CI presented below is about 150 lines long, and allows a user to either permanently or temporarily load other programs, and to execute them.

Below is an overview of the sample program's structure; the complete source text is at the end of this chapter. Only a few points need to be made beyond the comments in the program itself.

1. **STARTUP** is nominally a user program, which means it runs on the 68000's user-mode stack and uses the USP register to address data on the user stack.
2. But a CI which loads and executes other programs must be run on the supervisor-mode stack. It gets there by calling **CI_SWITCH** prior to doing anything useful. There is a limited amount of space on the supervisor stack, and the CI must not overflow that area. This is checked in a static way at the beginning of **OUTERINTERPRETER** in the example. In Pascal 2.0 a generous area of 5000 bytes is allowed for the CI stack. Remember however that interrupt service routines also run on that stack. A good rule is for the CI to occupy no more than a couple of thousand bytes of stack at worst.
3. Once a CI gets onto the supervisor stack, it can't exit! It must never terminate until the system is re-booted. If you write a CI which exits back to the system kernel, errors may or may not be reported, but the behavior will be unreliable.
4. The assembly language routine **USERPROGRAM** causes a loaded program to be invoked as a co-routine to the CI. It looks like a subroutine call syntactically, but **USERPROGRAM** first switches the CPU into user mode and thus switches to the user program stack. **USERPROGRAM** surrounds the user program with a **TRY-RECOVER**, then calls the start address as if it were a procedure.
5. The example includes some "magic code" which searches the loader's data structures to find out if a requested program is already in memory. It isn't important to understand why this code works, as long as you don't alter it. Just believe, there is a Santa Claus.

Structure of the Command Interpreter

program customsystem

A command interpreter is written and compiled as a user program which uses modules of the underlying kernel, such as the loader and file system. This program must be stored as file STARTUP on the boot device.

procedure outerinterpreter

The "outer interpreter" is a layer imposed mainly to intercept and report errors which occur when programs are run. Programs are invoked in response to user commands, which are processed by the "executive" below.

procedure cleanup

Error recovery is relatively simple. The command interpreter must reset certain system state variables.

procedure disableuserisrs

The user program may have installed its own interrupt service routines. Any ISRs which are temporary must be "disconnected" in case of error terminations, since the code itself may be gone after the program terminates.

procedure executive

This routine prompts the user for commands, and satisfies them by loading and/or executing program files. The ability to load files without executing them is the "induction rule" by which the system grows. Most other significant actions would be performed by running programs.

procedure go

Executes a program which has been loaded into memory.

procedure findorload

Searches for a program identified by its file name. If the program is already present in memory, its entry point is returned; otherwise an attempt is made to load the file.

begin executive

Here is where the human interface work is done.

end executive

begin outerinterpreter

Here is where errors are caught and reported, including errors which take place when loading program files.

end outerinterpreter

begin customsystem

The command interpreter first gets onto the 68000's supervisor stack, then executes the outer interpreter.

end customsystem

Sample Command Interpreter

```
$sysprog$
program customsystem (input,output);

{**** $search 'INTERFACE' or wherever you keep the interface spec }
import
asm,sysglobals,misc,ldr,loader,fs;

procedure outerinterpreter;
var
marker: anyptr;           {used to measure stack height}

procedure cleanup;       {called after error terminations}
var iu: 1..maxunit;
begin
locklevel := 0;          {unlock STOP key}
actionspending := 0;     {flush immediate-action keys}
for iu := 1 to maxunit do {force volume directory cleanups}
unitable^[iu].umediavalid := false;
end; {cleanup}

procedure disableuserisrs;
begin
call(cleariohook);      {clear all interfaces}
interrupttable := perminttable; {restore "permanent" ISRs}
end; {disableuserisrs}

procedure executive;
var
whatfile: fid;          {name of a program to execute}
ch: char;               {read user commands}

procedure go;           {execute a loaded program}
var
modpctr: moddescptr;   {step thru module descriptors}
allexecuted: boolean;  {files may have >1 executable module}
userheap: anyptr;      {used to clean up heap}
saveescape,            {keep user program's escape code}
saveior: integer;      {keep user program's IO error code}
begin
modpctr := entrypoint; {^desc of 1st module in loaded file}
repeat
{examine each module in the file}
allexecuted := modpctr^.lastmodule; {last one?}
if modpctr^.startaddr <> 0 then {module is a main program}
begin
mark(userheap);          {remember top of heap}
userprogram(modpctr^.startaddr,userstack); {do it!}
saveescape := escapecode; {how did it terminate?}
saveior := ioreult;      {may be an IO error}
release(userheap);      {retrieve heap space}
openfiles;              {re-open standard input & output files}
if saveescape <> 0 then {bad termination}
begin
allexecuted := true;    {don't execute any more}
disableuserisrs; {restore system interrupt routines}
end;
end;
end;
end;
```

```

        cleanup;           {force disc directory cleanups}
        if (saveescape<>-1)           {error but no message}
        and (saveescape<>-20)           {stop key}
        then
            printerror(saveescape,saveior);   {report facts}
        end;
    end;
    modptr := modptr^.link;           {look at next module in file}
until allexecuted;
end; {go}

procedure findorload;           {look for loaded program or load it}
label 1;
var
    vol: vid; name: fid; segs: integer; kind: filekind;
    modp: moddescptr; upcname: tid;
begin
    if scantitle(whatfile,vol,name,segs,kind)   {take apart name}
    then                                         {file name looks legitimate}
        begin
            if strlen(name)<=tidleng   {proposed name acceptable size?}
            then upcname := name       {will want uppercase version}
            else upcname := '';        {else don't bother}
            upc(upcname);               {upper case it}
            modp := sysdefs;           {root of list of resident modules}
            while modp <> nil do with modp^ do
                begin {search resident modules for requested file}
                    if startaddr <> 0 then           {module is a program}
                    if (name=programe)           {name user typed?}
                    or (ucase and (upcname=programe)) {try uppercase}
                    then                           {eureka! - we found it}
                        begin
                            if entrypoint<>modp   {most recently loaded one?}
                            then releaseuser;   {no, try to recover space}
                            entrypoint := modp;   {remember what we found}
                            goto 1;               {well-structured GOTO}
                        end;
                            modp := link;         {not this, look at next module}
                        end;
                    end;
                load(whatfile,false);           {not found - nonpermanent file load}
1: end; {findorload}

begin {executive}
    writeln(output);                               {prompt user}
    writeln(output,'Do you want to');
    writeln(output,'    Execute a program (type E)');
    write (output,'    Load a program (type L) ? ');
    read(input,ch); writeln(output);
    if (ch>='a') and (ch<='z') then                 {must upper case key}
        ch := chr( ord(ch)-ord('a')+ord('A') );
    if ch in ['E','L'] then                         {legitimate menu selection?}
        begin
            write (output,'What is the name of the program? ');
            readln (input,whatfile);
            fixname(whatfile,codefile);           {append '.CODE' if necessary}

```

```

    if ch = 'E' then
        begin
            findorload;                {make sure file is in memory}
            if entrypoint <> nil then go;    {if found, run it}
        end
    else
        begin
            load(whatfile,true);        {permanent load}
            markuser;                    {permanently increment heap limit}
        end;
    end;
end; {executive}

begin {outerinterpreter}
mark(marker);                {did this program take too much stack?}
if ord(marker) > userstack then escape(-2);    {out of memory}
markuser;                    {MUST MAKE COMMAND INTERPRETERS PERMANENT!!}
repeat {2 levels of repeat in case stop key is hit during cleanup}
    try                        {catch all errors}
        repeat                {repeat user prompt after each operation}
            try                {catch all errors}
                executive;        {ask what to do, then do it}
            recover            {if user program blows away ...}
                if escapecode <> -1 then {genuine error or simply halt?}
                    begin        {program had a problem (maybe stop key)}
                        cleanup;    {repair system variables}
                        disableuserisrs;    {repair interrupt structure}
                        printerror(escapecode,ioreresult);    {report problem}
                    end;
                until false; {CANNOT GRACEFULLY EXIT AFTER CALLING CI_SWITCH}
            recover            {errors or stop key during error recovery}
                printerror(escapecode,ioreresult);    {report them too}
        until false;
    end;
end; {outerinterpreter}

begin {customsystem}
    ci_switch;

    { NOW ON SUPERVISOR STACK. THIS PROGRAM CANNOT GRACEFULLY
      EXIT NOW. IT SHOULD NOT EVEN TRY TO DO SO! }

    outerinterpreter;        {execute the custom shell}
end. {customsystem}

```

Chapter 11

CPU Interrupt Handling

Introduction

This chapter discusses how the Pascal system sets up and processes interrupts. Although the section begins with a brief refresher on how interrupts work, you really need to know the material in the "MC68000 User's Manual".

Note

The topic covered here is the lowest level interrupt structure, not interrupts as handled by the device IO library procedures. That is covered in another section. Also, the system designers took pains to provide an easy-to-use special case for intercepting keyboard interrupts, which is discussed in the next chapter.

Interrupts are a special case of exception processing. The 68000 microprocessor has eight interrupt priority levels. Level zero means no interrupt active; levels one through six are the "maskable" interrupts which are used for interaction with peripheral interfaces. Level seven is the "non-maskable" interrupt NMI, which can never be disabled. NMI is not the same as RESET, which is a separate line into the CPU.

In the 9836, some of these levels are already consumed by built-in peripherals:

- Level 1 -- keyboard, knob and clock.
- Level 2 -- the two minifloppy drives.
- Level 3 -- internal HPIB port; also DMA card.
- Level 7 -- powerfail interrupt if present; also shift-PAUSE cntrl-shift-PAUSE.

The CPU has two states, called "user" and "supervisor". There is a separate copy of register A7 (the stack pointer) for each state. The user stack pointer is commonly denoted USP, and the supervisor one SSP.

An interface requests interrupt service over three signals (IPL0', IPL1', IPL2') coming into the CPU. The interface puts on these lines an octal number indicating the priority at which its request should be serviced. This value is noticed by the CPU between instructions (but note that NMI is handled slightly differently). The processor compares the priority at which it is currently operating, as designated in its status register, to the value on the IPL' lines. If the requested level is greater than the CPU level, an interrupt is granted. The processor pushes its state (the status register and program counter values) on the stack pointed to by SSP.

Then an Interrupt Service Routine (an ISR) is chosen for the interrupt by selecting an "interrupt vector". This can happen in two ways: a fixed location in memory (corresponding uniquely to the priority level) can provide the address of the service routine, or the interface itself can push a vector number onto the bus during an Interrupt Acknowledge operation. In either case, the processor derives from this "vector" the address, somewhere near address zero of memory, of a pointer to the service routine. Although both methods are possible with the 9836, as of this writing only the first method is actually used by HP interface.

In the 9836, all the exception vectors are in the Boot ROM. Each vector contains a pointer to a 6-byte area in RAM near \$FFFFFF. The intent is for a software system to put JMP instructions in these RAM areas, leading to the service routines. The Boot ROM itself puts initial values in, which lead to error reporting routines within the Boot ROM. For more details about the addresses of these RAM vectors and how they are initialized when the machine powers up, see the section on the Boot ROM.

Pascal control begins at the assembly language routine POWERUP, mentioned in the previous discussion of the Pascal boot process. As soon as POWERUP has created the Pascal proto-environment (stack and heap, plus error recovery block), it fills in all the RAM exception vectors. In particular, the locations for level one through six interrupts are set up as jumps to a single routine called INTERRUPT, within the POWERUP module itself. It is the duty of this routine to "interface" between all actual interrupts and the intended ISRs. The primary reason for this intervention is that we wanted to be able to write ISRs in Pascal, and there is no guarantee that the machine will be in a valid "Pascally" state when an interrupt occurs.

The state information saved includes all the registers, the current value of IORESULT, and the current value of the error variable ESCAPECODE. This is enough to assure that side effects of an ISR which does IO or gets errors will not foul up the running program. Additionally, a TRY-RECOVER is set up around the interrupt servicing itself. Any errors which occur and try to escape out of the ISR will be trapped and thrown away. (The STOP key, ESCAPECODE=20, is the only exception.)

A certain amount of interrupt servicing speed is lost in this mechanism, which is the price of convenience. It is claimed, by those who should know, that Pascal can service about 4000 interrupts per second through this mechanism. If that is a problem, there is the option of writing an ISR in assembly language and simply putting a jump to it into the appropriate RAM vector location. However, the vector MUST have its normal content if the Pascal IO subsystem is to deal with the interrupts. If you every try changing the vectors, be sure to put them right before your program exits back to the OS. Use a TRY-RECOVER statement surrounding the body of your program to be sure that error terminations won't deprive you of the opportunity to make the system honest again.

Interrupt servicing for a given priority level is complicated by the fact that there may be several interfaces which can all interrupt at that level. So there is a general mechanism for "chaining" together ISRs for the various interfaces which are operated on each priority level. This chain of ISR descriptions is searched by a process called polling, performed by the all-purpose service routine INTERRUPT. The best overview of polling is gotten by examining the Pascal descriptions. The polling routine simply interprets these Pascal structures.

In module SYSGLOBALS:

```
type
  pisrib = ^isrib;
  isrproctype = procedure (isribptr: pisrib);

  isrib = packed record
    intregaddr: charptr;
    intregmask: byte;
    intregvalue: byte;
    chainflag: boolean;
    proc: isrproctype;
    link: pisrib;
  end;
  inttabletype = array [1..7] of pisrib;

var
  interrupttable: inttabletype;
  perminttable: inttabletype;
```

In module ISR:

```
procedure isrlink (procentry: isrproctype;
  lintregaddr: charptr;
  lintregmask: byte;
  lintregvalue: byte;
  lintlevel: byte;
  isribp: pisrib);

procedure permisrlink (procentry: isrproctype;
  lintregaddr: charptr;
  lintregmask: byte;
  lintregvalue: byte;
  lintlevel: byte;
  isribp: pisrib);

procedure isrunlink (lintlevel: byte;
  isribp: pisrib);

procedure isrchange (procentry: isrproctype;
  isribp: pisrib);
```

In module POWERUP: (although their link-time names look like they're in ASM)

```
procedure setintlevel (level: integer);

function intlevel: integer;
```

Every ISR is a procedure which takes, as its single parameter, a pointer to an ISR information block (ISRIB).

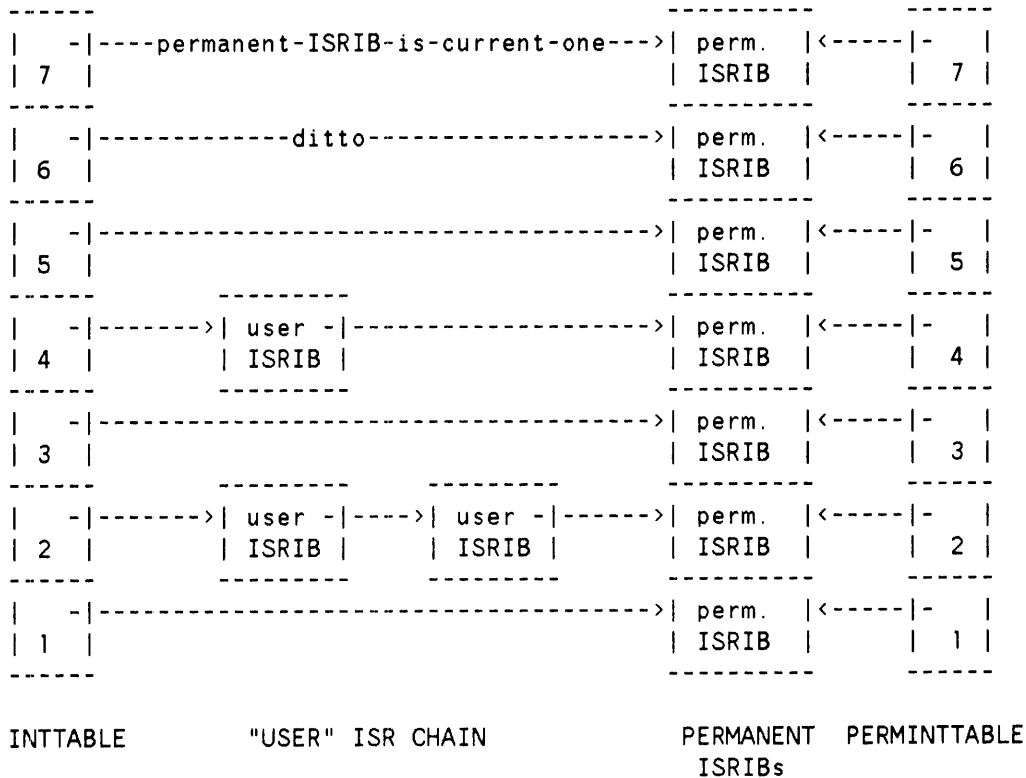
The ISRIB gives all the information needed to test if any single interface is interrupting. INTREGADDR is a pointer into the IO portion of address space; it accesses a one-byte register

indicating whether the interface is requesting interrupt service. This byte is ANDed with the INTREGMASK field, and the result is compared with field INTREGVALUE. If the two values are equal, the service routine PROC will be called, passing the address of the ISRIB itself.

(Recall from the system memory map presented in an earlier section that each IO select code is allocated 65536 bytes of address space. A trivium: it is usually the case that the byte-wide registers of our cards are placed in the odd byte of a word address. At byte offset 1 (ie the 2nd byte) from the low address end of the 65k allocated to a select code, one finds the card ID byte, which identifies the interface type. The "requesting interrupt" register which must be polled is probably at byte offset 3. Certain built-in peripherals whose IO mapping addresses are fixed, do not identify themselves.)

The other fields of an ISRIB have to do with polling, the search by which this test process is applied to all devices on a given priority level.

ISRs are accessed through the array called INTERRUPTTABLE. There are two classes of ISR: "permanent" and "user". Permanent ISR's are the ones the system expects to be present. They are NOT removed automatically whenever a program terminates, whereas user ISR's are removed then. User ISRs are simply temporary ones stacked at the front of the list or removed by calls to ISRLINK or ISRUNLINK. (ISRUNLINK will work on permanent ISRs too.)



ISR cleanup when a program terminates consists mainly of copying the values of PERMINTTABLE into INTERRUPTTABLE. The picture above does not depict the fact PERMINTTABLE may point to an ISRIB which is not the last one in the chain. That is, there may be several permanent routines for a priority level.

When an interrupt is granted, the INTERRUPT interfacing routine goes to the list of ISRIBs headed by INTERRUPTTABLE[INTLEVEL] and examines the first ISRIB, using the register

mask-and-compare procedure described above. In many cases, the "current" head of the list will simply be the permanent ISRIB installed by the Pascal OS. When a match is found, the PROC in the ISRIB is called.

This procedure will usually perform the requisite service, but need not necessarily do so; it may look at the device's status and choose to pass the interrupt on down the ISRIB chain. If it does NOT to service the interrupt, it must set the CHAIN field of the ISRIB to true (it was initially false); this causes polling to go on to the next ISRIB. If instead the procedure elects to service the interrupt, it must act on the interface so the interrupt is cleared.

If the end of the chain is reached and no one serviced the interrupt, the polling routine will restore the CPU to its state prior to interruption and execute a ReTurn from Exception (RTE) instruction. This lowers the CPU priority level, and the unserved interface will immediately interrupt again, producing an infinite loop. (By the way, there is an easy way to induce this undesirable behavior. It is not illegal for an INTTABLE entry to be nil; this is detected, and a dummy routine NOISR, exported from INITUNITS, is called. Unfortunately, NOISR can't clear the interrupt, so the system will just hang.)

There is no rule governing what action must be performed to "clear the interrupt condition" for an interface. Obviously resetting it will do so, but that isn't what you want! Some cards drop interrupt request when their data is read; others must be commanded explicitly through a control register.

You should be aware of two problems which can occur in ISR design: the "destructive read" and the "phantom interrupt".

Some interface circuits, notably the TI9914 HPIB chip, have the property that when their status is read, the status register is cleared. The circuit forgets its status! If different interrupt conditions are to be serviced by different interrupt routines, the status read by one ISR must be shared by all. Moreover, reading status may clear the interrupt request, so that the proper ISR won't get called at all. The solution is that every ISR must be prepared to invoke the others as "fake" interrupts if the interface status which was read requires this.

Phantom interrupts are a related problem. Suppose the CPU is running at level zero and (destructively) reads a status register, the clearing of which also drops the interrupt request. Things may happen in this sequence:

1. CPU starts instruction which reads the status register.
2. During execution of the read instruction, the interface for some reason raises its interrupt request.
3. The CPU captures data indicating the interface wants service. The instruction completes, and the interface -- having been read -- drops its interrupt request.
4. The 68000 only performs interrupts between instructions. As soon as the read completes, the CPU acknowledges the now rescinded interrupt request by spinning off into the ISR polling process.
5. But since the interface was cleared, no ISR finds anything to do.
6. Finally, execution resumes with the instruction following the unsynchronized read, and the level zero instruction sequence gets to provide the service which would otherwise have been done by the ISR.

Hooking in Your Own ISR

Your own ISR for an interface can easily be hooked into the system using the ISRLINK, ISRUNLINK and ISRCHANGE routines. You need to import SYSGLOBALS and ISR to do this.

See the restrictions on ISR procedures, below.

Nota Bene: this is different from writing ISRs to work as a part of the Pascal device IO library. The procedure to be described now bypasses the IO subsystem by coming in at the next-to-lowest possible level. (The lowest level would be to replace the RAM vector for the interrupt level.)

To install a new ISR at the head of a list, you will need to provide both the name of a service procedure taking a single ISRIBP as a value parameter (ie compatible with type ISRPROCTYPE), and the address of an ISRIB allocated in "safe" storage. Of course, you also need to specify the address of the "interrupt request" register, the mask and the target value for the polling match. All this is needed even if the ISRIB will be the only one at the chosen priority level.

Safe storage is storage whose lifetime is at least as long as the ISR is expected to be enabled. This means the ISRIB whose address you pass should either be a global variable, or taken from the heap in such a way that it will hang around as long as it is needed. (We tend to take them from the heap at INITLIB time.) It should NOT be a local variable of the procedure which sets up the ISR, UNLESS the ISR will be unlinked before that procedure exits!

To get storage from the heap, you can of course use the standard Pascal procedure NEW. If you want to use a global variable MYISRIB, you will need to pass its address "ADDR(MYISRIB)" since the link/unlink routines want a pointer to an ISRIB.

Call ISRLINK with the appropriate parameters to hook in a new ISRIB at the front of the INTTABLE list.

Call PERMISRLINK to hook in a new ISRIB at the front of the part of the list reached via PERMINTTABLE.

Call ISRUNLINK to remove any ISRIB from its priority chain, whether user or permanent.

Call CHANGEISR to change the name of the service routine represented by any one ISRIB, whether user or permanent.

A Cautionary Note

If you link in or change a permanent ISR, the system won't undo it. Don't use this feature unless you expect the system as a whole to run properly **WITH** your ISR in place. To replace an ISR during execution of a program, try to take advantage of the user ISR chain.

There are occasions when one wishes to leave new permanent ISRs around, for instance if they service several programs whose executions are chained together. In this case, you may wish to be sure of getting a chance to set things right in one of these programs should it bomb. A useful technique is to surround the main body of the program with a TRY-RECOVER statement:

```
:
begin (*main program*)
  try
    :
    : (* body... install user ISRs and use them *)
    :
    :
    escape(0); (*non-error branch into RECOVER code*)
  recover
  begin
    : (*undo ISRIB changes as necessary*)
  end;
end. (*main program*)
```

This technique assures that under all conditions, the ISR structure can be returned to normal by the programmer, rather than letting the system do its thing.

Restrictions on Interrupt Service Routines

There are a few significant restrictions on the routines you supply as ISRs. These are UNENFORCED rules; if your procedures violate them, the system may or may not be damaged, and you may or may not find out.

Error Conditions "Thrown Away"

The ISR interface routine, INTERRUPT, protects the system from errors during ISR execution by:

1. Saving the values of ESCAPECODE and IORESULT as part of the machine state.
2. Surrounding the ISR with a TRY-RECOVER to trap all errors which might otherwise try to escape out of the ISR.
3. Restoring the interrupted values of ESCAPECODE and IORESULT at the end of interrupt service.

The net result is that any errors which are detected during interrupt service are "thrown away". The sole exception is the STOP key. You should turn off all the checks enabled by Compiler directives when compiling ISR code, such as \$STACKCHECK\$, \$RANGE\$, \$OVFLCHECK\$, \$DEBUG\$.

The reason for this policy should be fairly apparent. If an ISR could surprise a program at any time by changing the values of IORESULT or ESCAPECODE, then attempts to structure program response to errors would be fairly futile. Note, however, that you may utilize TRY-RECOVER within an ISR. In this case, explicit programmed calls to ESCAPE are probably the only cause for recovery branches.

The "ISR in an ISR" Mistake

We have seen several cases of the following silly (but not unreasonable) mistake.

A program intercepts keyboard interrupts, which always arrive with priority level one. Then in the service routine, it tries to do keyboard IO: reading a character, reading the clock, etc. Unfortunately, the keyboard usually communicates with the 68000 by interrupting it, and since the request comes from within keyboard interrupt service, the CPU is already running at level one. The keyboard can't again interrupt with its reply until the current ISR exits, so the machine hangs up in the first ISR.

There is potential for this problem to appear elsewhere, especially in HPIB service since the TI9914 chip often communicates by interrupt.

Notes

Chapter 12

The Keyboard

Introduction

This chapter introduces the keyboard, a surprisingly complicated subject. The keyboard includes its own microprocessor which not only handles keystrokes but also manages the "knob" and maintains several timers capable of interrupting the CPU. There are at present two types of keyboards, 71 keys (small) and 105 keys (large), distinguished by the configurations jumpers on the keyboard. At present the small keyboard is detached from the mainframe while the large keyboard is attached. Associated 68000 system software takes care of communicating with the keyboard processor, mapping keystrokes into character values, handling timer interrupts and so forth.

In designing the Pascal software, we tried to anticipate some commonly needed extensions or alterations to the standard keyboard drivers, and to make it easy for programmers to achieve the desired behavior without getting involved in very low-level details. Many programmers -- perhaps most -- will be able to take advantage of these "hooks" and solve their problems expeditiously and with little effort. These time-saving features are discussed before getting into the gritty details of talking to the keyboard processor. You should read this higher-level material even if you presently suspect you "must" handle all the keyboard details yourself. The gritty stuff is covered afterward.

Summary of Keyboard Capabilities

Real time clock functions:

- Set time of day and date.
- Maintain time of day.
- Sample time of day.
- Set up real time match.
- Cancel real time match.
- Generate real time match interrupt.
- Set up delayed interrupt.
- Generate delayed interrupt.
- Cancel delayed interrupt.
- Generate periodic interrupt.

Non-maskable timeout:

- Set up delayed non-maskable timeout interrupt.
- Cancel non-maskable timeout.
- Generated non-maskable timeout.

10 msec periodic system interrupt (PSI)

- Enable or mask out the PSI

Beeper (tone generator) functions:

- Beep with specified frequency and duration.

Keyboard:

- Keycodes: 102 key matrix positions (large keyboard), qualified by SHIFT and CONTROL keys. 71 keys on small keyboard.
- Scanning: 2-key rollover with trailing-edge debounce (not available on the small keyboard).
- Interrupt 68000 on keystroke.
- Auto-repeat depressed key.
- Set auto-repeat rate.
- Set auto-repeat delay before first repetition.

Rotary pulse generator (the RPG or "knob")

- Detect rotation direction; 120 pulses per 360 degrees.
- Accumulate pulse count during specified period.
- Set up knob interrupts.
- Read knob pulse count.

Keyboard Access with the File System

The keyboard looks like an unblocked (byte stream) volume which can be read as a textfile using standard Pascal techniques. Since Pascal is a sequential language, a "typeahead" buffer retains keystrokes until they are read. You can observe this by pressing PAUSE and then typing on the keyboard. Hold down CNTRL and press CLR LN to clear the typeahead buffer.

The keyboard, like all HP Pascal text files, exhibits "lazy" IO behavior. This was discussed in the earlier section on the file system; it simply means that a character isn't read until it is needed. Nevertheless the file window F^ is always valid, meeting the ISO Pascal specification; the system does this by forcing a read at the time the window is accessed, if the window is not already valid due to a previous reference.

Echoing Read

The standard text file INPUT reads a character, string, or whatever from the keyboard. Each character is echoed to the CRT at the current cursor position. The precise rules of editing depend on the type of data being read. A single character is passed through unedited. A string is terminated by the ENTER key or the limit of the string's length, whichever happens first. You can NOT backspace from beyond the string's limit, back into a valid part of the string. A packed array of characters (PAC) is treated like a string, except it is filled with trailing blanks if necessary. Numbers are parsed according to a relaxed Pascal syntax, and integers are coerced into real numbers if necessary.

EOLN(INPUT) is true if the next character to be read corresponds to end-of-line (the ENTER key). If S is a string which is long enough to contain the rest of the line, then READ(INPUT,S) will leave EOLN true and the next character read will swallow the end-of-line and return a space (blank) character. Correspondingly, READLN(INPUT,S) will leave EOLN false and consume the end-of-line blank.

You can easily find out the ordinal value of the character returned by any keystroke, by running either of the programs below. Turning the knob will also return characters:

counterclockwise	#8 (ASCII backspace)
clockwise	#28 (ASCII Form Separator)
shift counterclockwise	#31 (ASCII Unit Separator)
shift clockwise	#10 (ASCII Line Feed)

In you try either sample program, you will discover that some keys beep rather than returning a value. We will soon discuss how you can dynamically alter the character mapping.

Non-Echoing Read

The same capability is accessible without the incoming keystrokes being echoed, using a different file than INPUT. In fact there are two ways.

Using the standard file KEYBOARD:

```
program readquietly (keyboard,output);
var
  ch: char;
  keyboard: text;
begin
  repeat
    read(keyboard,ch);
    write(output,'You pressed character #',ord(ch):1);
    if ch >= #32 then {printable character}
      write(output,' ',ch,' ');
    writeln(output);
  until false;
end.
```

Because KEYBOARD is a file passed in from "outside" the program, you must not RESET it. Like INPUT, it comes to you already connected to the proper logical unit. Nonetheless it must be declared both in the program heading and as a variable. For some reason, the HP Pascal language standard requires that only the standard files INPUT and OUTPUT must not be declared, but no files passed in as program parameters need be explicitly reset.

Connecting any text file to the keyboard:

```
program readquietly (output);
var
  ch: char;
  f: text;
begin
  reset(f,'#2:'); {open to logical unit one}
  repeat
    read(f,ch);
    write(output,'You pressed character #',ord(ch):1);
    if ch >= #32 then {printable character}
      write(output,' ',ch,' ');
    writeln(output);
  until false;
end.
```

There is a minor difference between these techniques. If you pass in the system file "keyboard", the FIB used will be the one which is always present for use by the OS. If you declare your own, a FIB (about 160 bytes) and a 512-byte buffer will be allocated. The 512 byte buffer has to be present, even though it isn't used for unblocked TEXT files, because the system doesn't know if the same file might not later be opened to mass storage instead of the keyboard. Despite this overhead, declaring the file yourself may be aesthetically preferable.

The Beeper

Although the beeper resides physically in the large keyboard and in the mainframe for the small keyboard, it can be triggered by writing an ASCII "bell" character to the standard file OUTPUT:

```
write(output,#G); { "bell" is control-G }
```

To control tone and duration, it is necessary command the keyboard directly, using system routines which are described later.

That pretty well covers what the keyboard can do using only the statements of standard Pascal, with no access to internal system routines. Next we discuss some of the system entry points which offer extended capabilities.

Easy-to-Use Extensions

Most of these capabilities are supplied by the modules KBD, KEYS, CRT, BAT and CLOCK which are part of the kernel. For now, ignore types, variables and procedures other than the ones of immediate interest.

Avoiding "Hanging Reads".

The nature of Pascal is that one statement must be completed before the next can begin execution. This goes for READ statements too. Pascal programs will wait forever if no one presses a key. This syndrome can be avoided by testing, before executing the READ statement, whether there are any characters waiting to be read in the typeahead buffer.

This test is performed by calling the boolean function UNITBUSY, passing the logical unit number corresponding to the keyboard. The keyboard is "busy" if the typeahead buffer is empty (waiting for at least one keystroke). The UNITBUSY predicate is exported from module UIO.

```
program avoidhangups (keyboard,output);
import uio;
var
  keyboard: text;
  ch: char;
  i: integer;
begin
  writeln(output,'Press any key.  Hurry!');
  for i := 1 to 300000 do
    begin
      if not unitbusy(2) then
        begin
          writeln(output,'You beat me!');
          read(keyboard,ch);
          halt;
        end;
    end;
  writeln(output,'Guess I was too fast for you.');
```

```
end.
```

Timing with the System Clock

You may wish to control the timing in the above example more accurately, by measuring elapsed time with the system clock. From `CLOCK` is exported an integer function `SYSCLOCK`, which returns the number of centiseconds (hundredths of a second) since midnight.

```
program avoidhangups (keyboard,output);
import uio,clock;
const
    timelimit = 200;           {200 centiseconds = 2 seconds}
var
    keyboard: text;
    ch: char;
    start: integer;
    gotone: boolean;
begin
    writeln(output,'Press any key within 2 seconds! ');
    start := sysclock; gotone := false;
    repeat
        if not unitbusy(2) then
            begin
                writeln(output,'You beat me!');
                read(keyboard,ch);
                gotone := true;
            end;
        until gotone or ((sysclock-start)>timelimit);
    if gotone then writeln(output,'Thank you.')
        else writeln(output,'#G,'Golly, you ARE slow!');
end.
```

Using the System Clock and Calender

`CLOCK` also can manipulate the calendar, and the clock in an hours, minutes, centiseconds format. The clock actually times in centiseconds since midnight; the hours-minutes-centiseconds format is just a convenience. `SYSGLOBALS` exports the following things:

```
type
    daterec = packed record
        year: 0..100;
        day: 0..31;
        month: 0..12;
    end;

    timerec = packed record
        hour: 0..23;
        minute: 0..59;
        centisecond: 0..5999; {per minute}
    end;

    datetimerec = packed record
        date: daterec;
        time: timerec;
    end;
```

CLOCK exports these things:

```
procedure sysdate (var thedate: daterec);
procedure setsysdate (thedata: daterec);
procedure systime (var thetime: timerec);
procedure setsystime (thetime: timerec);
```

To read the system date, declare a variable of type DATEREC and pass it to a call on SYSDATE. To set the system date, assign values to the fields of your DATEREC in the obvious way and pass it to a call on SETSYSDATE. If the computer stays on for more than 24 hours or has a powerfail device installed, the clock will retain time and roll over the date within ten minutes after midnight.

Similarly, a variable of type TIMEREC can be used to read or set the system clock in an hours, minutes, centiseconds format. The time is military time.

NB: none of these routines check to see that what is passed in makes sense!

Remapping the Keyboard

In reading the following material it may be helpful if you are familiar with interrupt servicing mechanism, which was presented in a previous section.

Sometimes it is necessary to redefine the mapping from key matrix positions to ASCII characters received by the file system. Remapping includes the capability to suppress keystrokes. Using the techniques presented now, it is possible to redefine all the keys except SHIFT, CTRL and RESET (SHIFT-PAUSE). SHIFT and CTRL are separated from all the other keys to allow for the computer to detect simultaneous depression of SHIFT or CNTRL and any other key. These two are "qualifiers" for the other keys. RESET is noticed as a special case by the keyboard processor and causes a Non-Maskable Interrupt instead of the usual level one service request. It cannot be remapped. However, it can be disabled by sending a low-level command directly to the keyboard processor. This is discussed with the gritty stuff.

First you need to understand a bit about how the keyboard microprocessor and the 68000 communicate. When the keyboard has something to say, it usually interrupts the 68000 on priority level one. Under certain circumstances it can force NMI (Non-Maskable Interrupt), but not for keystrokes. Conversely, the 68000 interrupts the keyboard processor when it wants to issue a command. The interrupts initiate or terminate communication; data is actually exchanged via memory-mapped IO, meaning certain locations in the 68000's address space form a bi-directional path between the processors instead of containing physical memory.

The detail you need now is that when the keyboard sends a message to the processor about a keystroke or anything else, the message is delivered as two bytes called Status and Data. For a keystroke, Data gives a value showing what key was depressed. By looking at Status the driver in the 68000 can determine what the nature of the message is, whether the SHIFT or CTRL keys were also depressed, and so forth.

When a keyboard interrupt is detected, the ISR reads Status and Data. If Status indicates a keystroke, the ISR must translate the keycode into a character, taking into account such factors as the language of the keyboard (Katakana, French, Spanish ...), whether the SHIFT key was down or CAPS LOCK active, and whether the key was an "immediate execute" function to be

performed by the ISR instead of passed to the file system. An example of immediate-execute is CTRL-BACKSPACE, which deletes the last character from the typeahead buffer.

The 8-bit character value which results from this translation is then normally stuffed into the typeahead buffer, from whence it will be eaten by the keyboard Transfer Method on behalf of the file system. If the buffer is full, the computer beeps and the keystroke is lost.

Knob interrupts are handled similarly, since Pascal translates them into ASCII characters, except that knob characters are only put in the typeahead buffer if it is empty. Otherwise they are discarded. So there can never be more than one character from the knob, and if there is one, it is at the head of the buffer. This prevents "inertia". For instance, the knob -- at 120 pulses per rotation -- can generate characters much faster than the Editor can scroll text vertically through the CRT. If the buffer filled up with scroll characters, it would be very hard to make the cursor stop where you want it.

In summary: the keyboard processor sends Status and Data to the 68000 for each character press. On the 68000 side, the ISR decodes Status and Data to derive an ASCII character which is stuffed into the typeahead buffer.

Here is What You Want to Know

The system's keyboard ISR provides an opportunity to examine and alter Status and Data before they are passed to the mapping algorithm. Thus you can change one keypress into another. You can also tell the system to NOT pass on the keystroke; then by recording it yourself in the typeahead buffer or in a private keyboard buffer, you can translate it as your heart desires.

This capability is provided by a procedure variable hook exported from KBD. An exactly analogous capability is provided for intercepting timer interrupts.

```
type
  kbdhooktype = procedure (var statbyte,databyte: byte;
                          var dokey: boolean);
  timerhooktype = procedure (var statbyte,databyte: byte;
                             var dokey: boolean);
var
  kbdhook: kbdhooktype;
  timerhook: timerhooktype;
```

The procedure named by KBDHOOK is called by the keyboard ISR whenever the keyboard interrupts with a keystroke or knob pulse count. The procedure activated by this call is of course actually run as part of the ISR -- it IS an ISR, and it must follow the guidelines on ISRs given previously. The call occurs sequentially right after Status and Data have been read, and before any Pascal-related processing takes place. (Unless the ANYCHAR key had been pressed and let through to the normal ISR processing; then the next three keypresses are handled by the system keyboard handler.)

KBDHOOK is initialized to the standard keyboard driver. When you assign the name of your own hook procedure, here is how it should interpret the parameters:

- STATBYTE is the value of the status register. Your routine can examine it, and since it is a VAR parameter it may also be altered.
- DATABYTE is the value of the data register. It also can be examined or altered.
- DOKEY is passed in set to TRUE. If set FALSE in the hook routine, no further processing of the interrupt will take place after the hook returns to the main ISR.

Exactly analogous remarks apply to timer interrupts.

KBDHOOK Status Byte

Here is the definition of the Status byte when KBDHOOK is called. Please note that the definition is different for other kinds of interrupts.

bit	meaning
7	Most significant bit always = 1 for KBDHOOK calls.
6	= 0 if keystroke, = 1 if knob pulse count.
5	= 0 if CTRL pressed, =1 if not.
4	= 0 if SHIFT pressed, =1 if not.
3-0	Should be ignored and not disturbed.

The simplest way to test one of these bits is with ODD and DIV. For instance, to set a boolean TRUE if CTRL was depressed,

```
control := not odd (status div 32);
```

Alternatively, you might figure out how to declare a packed record which has boolean fields in the right places and trick the value of Status into this record.

If Status indicates the interrupt came from the knob, the value of Data tells how many pulses have accumulated since the last knob interrupt. The pulse count is a signed byte, which is a representation the Compiler won't generate without some pranks on the part of the programmer. So the way to interpret the count is:

```
if Data < 128 then count := Data           {counter clockwise}
else count := 128-Data;                   {clockwise}
```

If bit 6 of Status = 0, indicating a keystroke, then the correspondence between Data values and physical keys pressed on a large keyboard is given by this picture. Please note that the "names" of the keys shown are those on the standard English keyboard. If your machine has a foreign or small keyboard, some of the keycaps may be in different positions. The physical position of the switch is what counts on the large keyboards.

The Large Keyboard

_	[k0][k1][k2][k3][k4]	[da][ua]	[il][dl][rc]	[ed][al][gr][st
/\	26 27 28 32 33	34 35	40 41 42	48 49 50 51
knb	[k5][k6][k7][k8][k9]	[la][ra]	[ic][dc][ce]	[cl][rs][pa][io]
_/	29 30 31 36 37	38 39	43 44 45	52 53 54 55

[]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[0]	[-]	[=]	[bs]	[ps]	[E]	[(]	[)]	[^]
	80	81	82	83	84	85	86	87	88	89	90	91	46	56	76	77	78	79

[tb]	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[]	[]	[rn]	[7]	[8]	[9]	[/]	
	25	104	105	106	107	108	109	110	111	100	101	92	93	47	72	73	74	75

[c1]	[]	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[ent]	[cn]	[4]	[5]	[6]	[*]
	24	112	113	114	115	116	117	118	102	103	94	95	57	58	68	69	70	71

[shift]	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[.]	[.]	[/]	[shift]	[1]	[2]	[3]	[-]
	120	121	122	123	124	125	119	96	97	98		64	65	66	67

[===== space bar =====]	[ex]	[0]	[.]	[.]	[+]
99	59	60	61	62	63

The Small Keyboard

Small keyboard must return keycodes that correspond to the equivalent keycap notation of the large keyboard since the physical arrangement of the small keyboard is not the same as the large keyboard. There are some keys on the large keyboard that are not available on the small keyboard due to the differences in the number of keys. In other cases, some keycaps of the small keyboard contain different pairs of functions than the large keyboard. Not all functions are shown on the keycap of the small keyboard and require the use of CTRL and SHIFT-CTRL to access the equivalent keycodes of the large keyboard. This is shown in the picture below of the small keyboard where (s) represents the status register indicating a shifted key, (c) indicating a control key and (&) indicating a shift/control key.

	[K5]	[K6]	[K7]	[K8]	[K9]	[1a]	[ra]	[rc]	[il]	[ic]	[dc]	[cl]	[st]	[io]
K ONLY	29	30	31	36	37	38	39	42	40	43	44	52	51	55
SHIFT	26	27	28	32	33	34	35	s42	41	s53	s54	s52	s51	s55
CTRL	s76	s77	s78	s79	s75	s71	s67	s63	s62	79	44	c52	c51	c55
SH/CT	s76	s77	s78	s79	s75	s71	s67	s63	s62	79	44	&52	&51	&55

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[0]	[-]	[=]	[bs]	[ps]
K ONLY	80	81	82	83	84	85	86	87	88	89	90	91	46	56
SHIFT	s80	s81	s82	s83	s84	s85	s86	s87	s88	s89	s90	s91	s46	rst
CTRL	c80	c81	c82	c83	c84	c85	c86	c87	c88	c89	c90	c91	c46	c56
SH/CT	&80	&81	&82	&83	&84	&85	&86	&87	&88	&89	&90	&91	&46	rst

	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]	[rn]
K ONLY	104	105	106	107	108	109	110	111	100	101	92	93	47
SHIFT	s104	s105	s106	s107	s108	s109	s110	s111	s100	s101	s92	s93	s47
CTRL	c104	c105	c106	c107	c108	c109	c110	c111	c100	c101	c92	c93	c47
SH/CT	&104	&105	&106	&107	&108	&109	&110	&111	&100	&101	&92	&93	&47

	[ct]	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[en]	[cn]
K ONLY		112	113	114	115	116	117	118	102	103	94	95	57	58
SHIFT		s112	s113	s114	s115	s116	s117	s118	s102	s103	s94	s95	54	s58
CTRL		c112	c113	c114	c115	c116	c117	c118	c102	c103	c94	c95	57	c58
SH/CT		&112	&113	&114	&115	&116	&117	&118	&102	&103	&94	&95	57	&58

	[shf]	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]	[shift]	[ex]
K ONLY		120	121	122	123	124	125	119	96	97	98		59
SHIFT		s120	s121	s122	s123	s124	s125	s119	s96	s97	s98		s59
CTRL		c120	c121	c122	c123	c124	c125	c119	c96	c97	c98		c59
SH/CT		&120	&121	&122	&123	&124	&125	&119	&96	&97	&98		&59

	[===== space bar =====]	[tb]	[c1]
	99	25	24
	s99	s25	s24
	c99	c25	c24
	&99	&25	&24

There is now a second-revision small keyboard. The keyboard can be identified by reading the keyboard configuration register. Bit 6 will be set if the keyboard is the second-revision keyboard. (Bit 6 is reset if the keyboard is the first-revision small keyboard.)

The second-revision small keyboard returns a different key-code for the SHIFT-CONTROL of most of the top row of keys. The ENTER key also returns a different key-code when SHIFT and/or CONTROL has been pressed.

Note

At this time, the small Swedish/Finnish keyboard (option 850) does not have the "*" character. The proposed location for this key is CONTROL DELETE CHARACTER. Depending on the revision of the keyboard, this key may be available.

	[K5]	[K6]	[K7]	[K8]	[K9]	[la]	[ra]	[rc]	[il]	[ic]	[dc]	[cl]	[st]	[io]
K ONLY	29	30	31	36	37	38	39	42	40	43	44	52	51	55
SHIFT	26	27	28	32	33	34	35	s42	41	s53	s54	s52	s51	s55
CTRL	s76	s77	s78	s79	s75	s71	s67	s63	s62	79	44	c52	c51	c55
SH/CT	s26	s27	s28	s32	s33	c38	c39	c42	c40	c43	c44	&52	&51	&55

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[0]	[-]	[=]	[bs]	[ps]
K ONLY	80	81	82	83	84	85	86	87	88	89	90	91	46	56
SHIFT	s80	s81	s82	s83	s84	s85	s86	s87	s88	s89	s90	s91	s46	rst
CTRL	c80	c81	c82	c83	c84	c85	c86	c87	c88	c89	c90	c91	c46	c56
SH/CT	&80	&81	&82	&83	&84	&85	&86	&87	&88	&89	&90	&91	&46	rst

	[Q]	[W]	[E]	[R]	[T]	[Y]	[U]	[I]	[O]	[P]	[[]	[]]	[rn]
K ONLY	104	105	106	107	108	109	110	111	100	101	92	93	47
SHIFT	s104	s105	s106	s107	s108	s109	s110	s111	s100	s101	s92	s93	s47
CTRL	c104	c105	c106	c107	c108	c109	c110	c111	c100	c101	c92	c93	c47
SH/CT	&104	&105	&106	&107	&108	&109	&110	&111	&100	&101	&92	&93	&47

	[ct]	[A]	[S]	[D]	[F]	[G]	[H]	[J]	[K]	[L]	[;]	[']	[en]	[cn]
K ONLY		112	113	114	115	116	117	118	102	103	94	95	57	58
SHIFT		s112	s113	s114	s115	s116	s117	s118	s102	s103	s94	s95	54	s58
CTRL		c112	c113	c114	c115	c116	c117	c118	c102	c103	c94	c95	c57	c58
SH/CT		&112	&113	&114	&115	&116	&117	&118	&102	&103	&94	&95	&57	&58

	[shf]	[Z]	[X]	[C]	[V]	[B]	[N]	[M]	[,]	[.]	[/]	[shift]	[ex]
K ONLY		120	121	122	123	124	125	119	96	97	98		59
SHIFT		s120	s121	s122	s123	s124	s125	s119	s96	s97	s98		s59
CTRL		c120	c121	c122	c123	c124	c125	c119	c96	c97	c98		c59
SH/CT		&120	&121	&122	&123	&124	&125	&119	&96	&97	&98		&59

	[===== space bar =====]	[tb]	[cl]
		99	25 24
		s99	s25 s24
		c99	c25 c24
		&99	&25 &24

An Example Keyboard Program

At this point an example may serve to bring together the information that has been presented so far. This program installs a keyboard hook which detects the "user keys" K0 through K9 and performs certain functions when the keys are pressed. It also illustrates certain good programming practices which should be followed by any program using KBDHOOK. When the program is running, the following key definitions obtain:

- **K0** displays the system time in centiseconds.
- **K1** starts a "stop watch" function.
- **K2** displays time since K1 was pressed last.
- **K3** displays the system time in military format.
- **K4** displays the system date in Month-Day-Year format.
- **K5-K8** display the number of the soft key depressed.
- **K9** stops the program and restores KBDHOOK.

The Program

```
$sysprog$    {Enables system programming features: TRY-RECOVER and
              procedure variables}

program samplesoftkeys (input,output);

import
    {These modules need to be in the system LIBRARY
     or found in the INTERFACE library via a $SEARCH
     compiler directive.}

    sysglobals, kbd, clock;

var
    savehook          : kbdhooktype;
    starttime         : integer;
    endtime           : real;
    time              : timerec;      {timerec and daterec are}
    date              : daterec;      {exported from KBD}
    done,
    control,
    shift,
    clockflag,
    startflag,        {type byte is exported from}
    endflag,          {module SYSGLOBALS}
    timeflag,
    dateflag,
    quitflag,
    unimplemented     : boolean;
    key               : string[15];
```

```

procedure softkeyhook (var status,data: byte; var doit: boolean);
begin
  if doit then          {Only process key if not previously taken}
                        {See explanatory remarks below.}
    if not (data in [26,27,28,29,30,31,32,33,36,37]) then

      call(savehook,data,status,doit)  {Pass to previous hook}
                                       {if not processed here}

    else      {it's a soft key}
      begin
        doit := false;          {makes KBD ISR ignore keypress}
        control := not odd(status div 32);
        shift := not odd(status div 16);

        case data of
          {K0} 26: clockflag := true;          {K0 tells the time}
          {K1} 27: startflag := true;         {K1 starts stop watch}
          {K2} 28: endflag := true;          {K2 stops the stop watch}
          {K3} 32: timeflag := true;         {K3 also tells time}
          {K4} 33: dateflag := true;         {K4 gives the date}

          {K5..K8} 29,30,31,36:
            begin
              unimplemented := true;
              key := '';
              if control then key := 'CTRL ';
              if shift then key := key+'SHIFT ';
              case data of
                {K5} 29: key := key+'K5';
                {K6} 30: key := key+'K6';
                {K7} 31: key := key+'K7';
                {K8} 36: key := key+'K8';
              end; {case}
            end;

          {K9} 37: quitflag := true;
        end; {case}
      end;
end; {softkeyhook}

begin {main program}
  try
    savehook := kbdhook;      {keep old hook, restore at end}
    kbdhook := softkeyhook;
    unimplemented := false;
    clockflag := false;
    startflag := false;
    endflag := false;
    timeflag := false;
    dateflag := false;
    quitflag := false;
    done := false;

    writeln('PRESS ANY SOFT KEY AND SEE WHAT HAPPENS. ');
    writeln(' TRY SHIFT AND CTRL, TOO. ');

```

```

repeat          {loop, testing flags and performing actions}
  if unimplemented then
    begin
      unimplemented := false;
      writeln('#G,key,' is not implemented. ');
    end;
  if clockflag then
    begin
      clockflag := false;
      writeln('Clock time is ',sysclock:1);
    end;
  if startflag then
    begin
      startflag := false;
      starttime := sysclock;
      writeln('Stopwatch reset and started. ');
    end;
  if endflag then
    begin
      endflag := false;
      endtime := (sysclock-starttime) / 100.0;
      writeln('Elapsed time is ',endtime:7:2,' seconds');
    end;
  if timeflag then
    begin
      timeflag := false;
      systime(time);
      writeln('Current military time is ',
              time.hour:1,':',
              time.minute:1,':',
              time.centisecond div 100:1);
    end;
  if dateflag then
    begin
      dateflag := false;
      sysdate(date);
      writeln('Today''s date is ',
              date.month:1,'/',
              date.day:1,'/19',
              date.year:1);
    end;
  if quitflag then
    begin
      kbdhook := savehook;      {normal exit, restore hook}
      done := true;
    end;
until done;

recover          {catch errors, STOP key etc.}
begin
  kbdhook := savehook;      {error exit restores hook too!}
  escape(escapecode);      {pass error on out to OS}
end;
end. {main program}

```

Commentary on the Example

Using `KBDHOOK` accomplishes a considerable modification of the system's behavior while avoiding the necessity of writing a complicated new ISR which does everything the standard one did and more. Instead we extended the system's keyboard ISR.

The main program begins by saving the current value of `KBDHOOK` in a variable of the program. This serves two purposes. The most important one is that the original value **MUST** be restored before the program exits, since `SOFTKEYHOOK` will vanish as soon as the program terminates. Notice how a `TRY-RECOVER` in the main program ensures that no error condition can cause the program to fail to restore the previous keyboard hook. Restoration occurs both at the end of normal flow through the `TRY`, or in the `RECOVER` clause in case of error or `STOP` key.

The second purpose is a sort of courtesy or protocol: keypresses which are not interesting to our particular key hook are passed on to the previous hook. If every keyboard hook follows this convention, it is perfectly possible to have a chain of hooks which together do some fancy dynamic mapping of the keyboard. At any rate, the standard (system) keyboard driver is usually the last hook in the chain.

Procedure `SOFTKEYHOOK` is the means of detecting soft key presses, which normally get a "beep" and no character. This example works by setting flags indicating what keys have been pressed. The main program just runs around in a loop testing and resetting the flags. This very simple strategy is well suited to asynchronous user keypresses. Other strategies, such as putting bytes into the typeahead buffer so they appear to `READ` statements, may be better for your needs.

`SOFTKEYHOOK` runs as an ISR, since it is called from inside the system keyboard ISR. It therefore has the following characteristics:

- Reasonably short and fast.
- Almost no stack space needed.
- Declared in the outer block (global scope) of program.

It might seem tempting to instead perform the operations done by the main program inside `SOFTKEYHOOK` itself. Unfortunately that leads to disaster, because the `SYSDATE`, `SYSTIME`, and `SYSCLOCK` procedures themselves try to talk to the keyboard processor. Since `SOFTKEYHOOK` is already running at priority level one as a service routine for a keypress, there is no way for the keyboard to signal the 68000 when the answer to a clock time query or other request is ready to be read. Instead, the system will hang up.

Note also the avoidance of IO such as `WRITELN` from within `SOFTKEYHOOK`. The main reason for this is to avoid the unknown stack usage of complicated routines. It is in fact possible to do file IO while at priority level one.

It may be desirable to have user keyboard hooks which are installed automatically whenever the computer boots up, and are in effect permanently (from one program to the next). This can be done by appending to `INITLIB` a program containing the user keyboard hook procedure and an assignment statement to install it.

Gritty Details of the Keyboard

The next several pages present an abbreviated "Internal Reference Specification" of the keyboard with emphasis on matters of interest to a programmer, including a description of all the commands the 68000 can issue to the keyboard processor. At the end of the section we return to the Pascal 2.0 KBD module, which exports a routine which can be used to command the keyboard directly.

About the Electronics

In the 9826 and 9836 the keyboard control circuits are on the "mother board", which also provides connectors for the CPU board, floppy and CRT controllers, backplane and some other miscellaneous stuff. In the 9816, the keyboard control circuits are on the CPU board where the scan circuits are on the keyboard.

Keyboard Microcomputer

Keyboard functions are performed by an 8041 single-chip microcomputer, which is particularly well adapted to the sorts of scanning and control functions needed.

Clock

The clock is a 10 MHz canned crystal oscillator. It provides the time base for the floppy disc controller, the CRT, built-in HP-IB port, and keyboard/realtime clock. The keyboard and HP-IB need 5 mHz and get it by dividing the main clock. The various timers of the "real-time clock" are mainly implemented by software in the 8041.

Non-Detachable Keyboard Scanning

The keyboard appears electrically as a matrix of thirteen columns by eight rows; thus 104 distinct keypresses can be recognized. The 8041 polls the keyboard by outputting a seven bit address selecting a key.

This address is split out so the upper four bits can be used as a column address and the bottom three as a row address. The four bit column address is demultiplexed into 16 individual output lines; thirteen actually drive the thirteen columns of the key matrix, while the remaining three are used for other control purposes. The three bit row address drives a one-of-eight data selector. The output of this selector goes into the 8041, which reads it as the instantaneous state of the key being addressed. When the 8041 interrupts the 68000 to report a keypress, the full seven bit address is given as the Data byte to identify the key. The exact mapping of key addresses to switch positions is shown by a picture in the immediately preceding section.

The rotary pulse generator (knob) drives a counter which is reset each time the keyboard is scanned. The counter is read like any bank of keyswitches, but the information received by the 8041 is used to update an internal count register within the 8041. If the knob is turned clockwise, the count read from the keyboard is subtracted from the current internal (8041) counter value. The internal RPG counter is not checked for under or over flow conditions. The direction of the RPG is detected in the keyboard and is read as a bit during the keyboard scan.

Circuits Common to both Keyboards

The RESET (SHIFT-PAUSE) key is scanned like any other key but handled specially by the 8041. Instead of outputting the keycode, the 8041 generates an NMI. When the 68000's NMI service routine tries to decide what caused the interrupt, it looks at bit two of the Status register. If zero, the interrupt indicates the RESET key (SHIFT-PAUSE) was pressed; if one, the interrupt is a timeout. Although the RESET key can't be remapped, its interrupting can be suppressed by a command to the keyboard processor discussed below.

Keystroke debounce occurs in the 8041 software. The leading edge (key going down) is only debounced for 0.1 msec, enough to prevent electrical noise from causing a false key. Mechanical debounce is on the trailing edge (key being released). A key is not considered gone until at least 3 periods of 10 msec have passed.

The Rotary Pulse Generator (Knob)

When the knob is turned, it pulses a pair of signal lines 120 times per revolution. These signals are about 90 degrees out of phase, so it is possible to tell which direction the knob is moving. In the case of the non-detached keyboard, the 8041 receives two derived signals; one says the knob has pulsed, the other tells which direction. The detached keyboard knob scan is discussed under the section dealing with the detached keyboard.

The Beeper

The beep frequency is controlled by a six bit latch. There are 63 possible tones; tone number zero turns off the speaker, while number 63 is the highest pitch. The actual beep frequency will be 81.38 times the frequency in the latch. Duration of beep is controlled by software in the 8041.

Protocol for Keyboard Handling

Communication Addresses

Status register:	read byte at address	\$428003
Command register:	write byte to address	\$428003
Data to 68000:	read byte at address	\$428001
Data to 8041:	write byte at address	\$428001

The Status register may be read at any time. Whenever the keyboard interrupts, its ISR must first read Status and then read Data. Data must be read even if Status indicates a condition which has no associated data, because the read operation is what clears the interrupt.

Interrupting the 68000

The keyboard can interrupt on either level one or level seven, depending on the type of interrupt. Most communication uses level one; NMI generated by the keyboard is reserved for timeout and to indicate that RESET (SHIFT-PAUSE) has been pressed. (There are other sources of NMI than the keyboard.)

When level seven interrupt is being requested, the keyboard processor sets an unused bit in one of the status registers of the built-in HP-IB port. This is certainly an odd place to look! It came about as a result of adding the feature at the last moment. Anyway, if bit one of the byte at \$478005 is a one, the keyboard is requesting NMI. There are other, non-keyboard sources of NMI, so you should check this. For our case, bit two of Status indicates the cause of interrupt: if zero, the interrupt indicates the RESET key (SHIFT-PAUSE) was pressed; if one, the interrupt is a timeout.

Bit zero of the keyboard Status register is a one when level one service is desired. The 8041 can be commanded to mask any or all its reasons for interrupting. When any capability of the 8041 is masked, it not only won't interrupt to request service for that capability; it also will not set bits in Status indicating the service is wanted.

Of course, masking the 8041 is different from masking interrupts in the 68000. If the 68000 is running at level one or higher, and the 8041 is enabled to interrupt, the Status register will so indicate. Status can be polled at any time.

Sending a Command to the 8041

To tell the 8041 to do something, the general scheme is:

- Wait until bit one of the Status register is zero.
- Write a one-byte command to the Status register.
- Some commands need from 1 to 3 bytes of data; for each byte:
 - a. Wait until bit one of Status is zero.
 - b. Write the data byte to the Data register.

Bit one is the 8041's side of the "handshake"; when it is zero, the 8041 is ready for the next byte of data or has carried out the command.

"Black Box" Description of Functions

The following descriptions refer to a thing called the "timer output buffer", which is simply 5 contiguous bytes in the 8041's 64 byte address space. This buffer can be examined by commands to the keyboard processor.

A command sent to the 8041 cannot be considered carried out until bit one of Status is cleared to zero. If 68000 code, following a command which masks interrupts or turns off a timer, depends on not getting an interrupt from the masked or cancelled function, it must wait for Status bit one to clear. The 8041 is considerably slower than the 68000, and it may be off servicing a timer when the command arrives. Verbum sap.

When any command is sent which asks for data, an interrupt will occur in response. This interrupt can't be masked.

Set Time-of-Day and Date

Command: \$AD (173 decimal)

Begins counting time from the moment the 8041 begins executing the command. Follow with 3 or 5 bytes of data, least significant byte (LSB) first. First 3 bytes are the number of centiseconds (10 msec) since midnight. Last 2 bytes, if sent, are an integer representing the "day" in whatever calendar you prefer.

Set Date

Command: \$AF (175 decimal)

Follow with 2 bytes, LSB first, just like the optional last two bytes of the \$AD command.

Maintain Time-of-Day

Not a command. The 8041 updates 5 bytes. When the 3 LSB, which count centiseconds since midnight as an integer, reach 23 hours, 59 minutes, 5999 centiseconds, the day (top two bytes) is incremented. Invalid times \geq 24 hours will be rolled over and become valid within 10 minutes.

Sample Time-of-Day

Command: \$31 (49 decimal)

Copies the 5 bytes of real time/date into the timer output buffer. Further commands are necessary to read it.

Read Timer Output Buffer

Commands: .suspend \$13 (19 decimal) read timer buffer's LSB \$14 (20 decimal) read 2nd timer output byte \$15 (21 decimal) read 3rd timer output byte \$16 (22 decimal) read 4th timer output byte \$17 (23 decimal) read 5th timer output byte

Five separate commands; each sends a byte to the Data register and interrupts the 68000. These interrupts are not maskable within the keyboard processor.

Set Real-Time Match

Command: \$B4 (180 decimal)

Takes 3 bytes of data, so match is within 24 hours. Cancels itself after it occurs. Sending an invalid time will result in no match unless the clock is also invalid. The 8041 will begin checking for the match after it receives the third byte. NB: if you set the time while a realtime match is active, spurious match interrupts may be generated.

Cancel Real-Time Match

Command: \$B4 (180 decimal)

Same as setting up a match, except no data bytes follow. Can be sent at any time. Erases any pending (but masked) interrupt. Does not release level one interrupt request if it is already asserted as the result of a real time match. After this command is accepted no interrupt will be generated as a result of a match, regardless of whether the match is masked and/or logged at the time of accepting the command.

It is advisable to cancel real-time match when the real time is about to be changed, as an erroneous match interrupt could occur.

Sample Match Time

Command: \$38 (56 decimal)

Copies the match time into the timer output buffer. It may be read by command sequence \$13, \$14, \$15.

Generate Match Interrupt

Not a command; goes along with the next Periodic System Interrupt if PSI is enabled, else timer interrupt is generated by itself.

Set Delayed Interrupt

Command: \$B7 (183 decimal)

Takes 3 bytes representing delay period in centiseconds, LSB first. Immediately cancels any pending delayed interrupt. Begins counting time after receipt of 3rd byte. NB: the time is sent complemented; 0=longest delay, all 1's is the shortest. The result of shortest could be anywhere between 0 and 10 msec.

Cancel Delayed Interrupt

Command: \$B7 (183 decimal)

This is just a Setup command followed by no data. Can be sent at any time. Erases any pending but masked interrupt, etc. -- See Cancel Real Time Match.

Sample Delay Timer

Command: \$3B (59 decimal)

Copies the three bytes of the current delay timer into the first three bytes of the timer output buffer. The LSB will be in the first byte of the buffer. Use the \$13, \$14, \$15 commands to read the result.

Generate Delayed Interrupt

Not a command. The interrupt rides along with the next PSI (if enabled), or happens by itself.

Set Cyclic Interrupt, Copy Cyclic Timer, Cancel

Commands:

```
$BA (186 decimal)  set up interrupt
$BA (186 decimal)  with no data: cancel
$3E (62 decimal)   copy to timer output buffer
```

These commands are analogous to the delayed interrupt commands.

Generate Cyclic Interrupt

Not a command. If no PSI is already pending, log a normal interrupt; if pending, increment the counter, saturating at 31.

Set up Non-Maskable Timeout

Command: \$B2 (178 decimal)

Takes two bytes, LSB first, indicating the number of centiseconds until timeout. The value is 2's complemented, so 0 = 633560 msec and 255 = 10 msec. This command cancels any pending non-maskable timeout immediately. Begins counting time after receipt of second byte.

The designation "non-maskable" refers to the fact that the interrupt generated is on level seven. This timeout is in fact maskable in that the 8041 can be told to keep it quiet for a while.

Cancel Non-Maskable Timeout

Command: \$B2 (178 decimal)

Analogous to Cancel Delayed Interrupt; send the setup command followed by no data. Also releases NMI if it was asserted. 68000 has to wait so many micro-seconds to be sure NMI was released, but nowhere can I find the value of "so many". It is probably less than 10 msec.

Generate Non-Maskable Timeout

Not a command. Set bit 2 of Status register, indicating "not RESET key". This flag bit won't be overwritten even if RESET is pressed before the 68000 reads the status register, because the RESET key is artificially delayed 1.6 msec from when it goes down. Assert NMI. NMI will remain asserted until either a system Reset or Cancel command occurs.

Beep

Command: \$A3 (163 decimal)

Takes 2 bytes of data. Duration byte is the complement of the length of time to beep (00000000 = 2560 msec; 11111111 = 10 msec). Frequency byte is ≤ 63 , where 0 means stop beeping and 63 is the highest frequency. The actual beep frequency will be 81.38 Hz times the value of the frequency byte.

Auto-Repeat

Feature: All keys auto-repeat except RESET, SHIFT, CONTROL. Repeating starts after a given key has been down longer than the specified delay.

Set Auto-Repeat Rate

Command: \$A2 (162 decimal)

Takes one data byte which is 2's complement of desired repeat rate. 0 = no repeat; 1 = 2550 msec; 255 = 10 msec.

Set Auto-Repeat Delay

Command: \$A0 (160 decimal)

Takes one data byte which is 2's complement of the required delay. 0 = 2560 msec; 255 = 10 msec.

Set Knob Pulse Accumulation Period

Command: \$A6 (166 decimal)

The knob accumulates pulses for a specified period before interrupting the 68000 to report the count. This command specifies the period with one data byte. The number sent is the number of 10 msec in the period, with a funny behavior for a period of zero. 1 = 10 msec; 255 = 2550msec; 0 = 2560 msec.

The knob is normally scanned every 100 us; 200us when a key is detected; 500 us when a 10 ms interrupt occurs.

Read Knob Counter

There is no command to do this; the knob must reach the end of its accumulation period and send the data via an interrupt.

Generate Periodic System Interrupt

Not a command. The keyboard will interrupt every 10 msec when PSI is not masked. PSI interrupts will occasionally be accompanied by real-time clock interrupts (ie bits in the Status register will be set). It is possible to get one more PSI after masking it. The jitter in the PSI interrupt could be as much as 2 msec.

Mask Interrupts

Command: 010xxxxx binary.

The five bits xxxxx each correspond to some excuse for the 8041 to interrupt. Sending a bit = 1 will suppress that interrupt when its condition arises; sending a zero will enable it.

- bit 0 Mask keyboard and knob.
- bit 1 Mask the RESET key.
- bit 2 Mask the timer interrupts.
- bit 3 Mask the 10 ms periodic system interrupt.
- bit 4 Mask the "non-maskable timer interrupt".

Read interrupt mask

Command: \$04 (4 decimal)

Sends the current interrupt mask back through the Data register.

Read Configuration Jumper

Command: \$11 (17 decimal)

Sends back a value indicating which jumper on the mother board configuration is closed:

- Non-Detached Keyboard

0	no jumper	5	J5
1	J1	6	J6
2	J2	7	J7
3	J3	8	J8
4	J4		

- Detached Keyboard

bit 0	J3	(0 for large keyboard, 1 for small keyboard)
bit 1	J4	(unassigned)
bit 2	J5	(unassigned)
bit 3	J6	(unassigned)
bit 4	J7	(unassigned)

Note

The unassigned jumpers are wired connected yielding a "0" in the associated bit position. The remaining bits 5 through 7 are set to "0".

Read Language Code

Command: \$12 (18 decimal)

Sends the keyboard language code back through the Data register. The values whose meanings have been assigned are

- Non-Detached Keyboard

0	standard English keyboard	5	Katakana
1	French	6	Jumper J8
2	German	7	Jumper J10
3	Swedish/Finnish	8	Jumper J11
4	Spanish		

- Detached Keyboard

The values listed above for the non-detached keyboard apply as well to the detached keyboard for values zero through 5. Jumper designation J8, J10, and J11 do not apply. Actually, the implementation on the detached keyboard consists of three jumpers, J0, J1, and J2. These are read during the power up or 68000 reset first scan of the keyboard and not read again. A cut jumper indicates as a "1", therefore a Spanish keyboard should have jumper J2 cut with J0 and J1 uncut or connected. The remaining jumpers J4 through J7 are used as configuration jumpers as discussed above.

Keyboard Command Processing

Naturally there's a pattern buried in the preceding description. All the commands that involve getting information from the keyboard/realtime clock are specific cases of just two commands. The 8041 has 64 bytes of on-board RAM, and every byte can be read by the 68000.

Send the command 000xxxxx to address any byte in the lower 32 bytes of 8041 memory (xxxxx is a number between zero and 32). The command will answer by an interrupt and the Data register.

Send the command 001xxxxx to cause five bytes from the upper 32 to be copied down into the timer output buffer, which can then be read by 000xxxxx. Note that 1xxxxx is the highest location to be loaded; the lowest location is 1xxxxx-4.

Now here is the use of every byte of 8041 memory:

\$0-\$1 Scratch

\$2 These bits are a set of flags:

- 0-2 used for debounce
- 3 =1 when the non-maskable timer is in use
- 4 =1 when the cyclic timer is in use
- 5 =1 when the delay timer is in use
- 6 =1 when the match timer is in use
- 7 =1 when the beeper is on

\$3 Matrix or scan address of the key switch being scanned

\$4 Another set of flag bits:

- 0 =1 when keyboard/knob are masked
- 1 =1 when RESET is masked
- 2 =1 when user timer interrupts are masked
- 3 =1 when 10 msec PSI is masked
- 4 =1 when "non-maskable timer" is masked
- 5 =1 when it is time to auto-repeat
- 6 =1 when large keyboard is connected (detached kb)
- 7 first true bit for auto-repeat (detached kb)

Note: bits 6 and 7 are not used with the non-detached keyboard.

\$5 More flag bits:

- 0 =0 when SHIFT key is down
- 1 =0 when CTRL key is down
- 2 =1 when it is time to do a PSI
- 3 =1 when time to do user timer interrupt
- 4 =1 when time to do a knob interrupt
- 5 =1 when time to send 68000 something it asked for
- 6 not used
- 7 =1 when time to do non-maskable timer interrupt

\$6 Roll-over key save buffer
 \$7 The current key that is down

 \$8-\$F The 8041's stack space

 \$10 Reset debounce counter
 \$11 Configuration jumper code
 \$12 Language jumper code

 \$13-\$17 The 5 bytes of timer output buffer. LSB is \$13.

 \$18-\$1A Scratch

 \$1B Timer status bits:
 0-4 The number of cycle interrupts missed
 5 =1 when a cycle is up
 6 =1 when a delay is up
 7 =1 when there was a real-time match

 \$1C Current knob pulse count
 \$1D Location to put data sent by 68000
 \$1E Location for data to send to 68000
 \$1F Six-counter for 8041 timer interrupt
 \$20 Time to wait to start auto-repeating
 \$21 Auto-repeat timer
 \$22 Auto-repeat rate
 \$23 Beep frequency
 \$24 Beep timer (counts up to zero)
 \$25 Knob timer
 \$26 Knob interrupt rate
 \$27 If bit 6 is a one, the 8041 timer has interrupted

 \$28-2C Not used

 \$2D-\$2F Three bytes of time-to-day; LSB is \$2D

 \$30-\$31 Two bytes of days integer; LSB is \$30

 \$32-\$33 Two bytes of "non-maskable timer". LSB is \$32.

 \$34-\$36 Three bytes of real time to match; LSB is \$34.

 \$37-\$39 Three bytes of delay timer; LSB is \$37

 \$3A-\$3C Three bytes of cycle timer; LSB is \$3A

 \$3D-\$3F Three bytes of cycle timer save; LSB is \$3D

Processing an 8041 Service Request

This stuff goes on at priority level one. Either the 8041 will have interrupted, or the request for service may have been noticed by polling bit zero of Status. In either case bit zero of Status is true.

- When the sequence starts, read Status to determine the nature of the service request.
- Always read Data once if there is no data associated with the service. This clears the interrupt. If N (greater than zero) bytes are expected from Data in response to a request, read Data EXACTLY N times. Reading too much or too little data can cause missed interrupts or misinterpretation of interrupts.
- For each subsequent byte of data that goes with the message, the 8041 will interrupt once. Your code needs to "know" how many data bytes to expect.
 - a. Wait for bit one of Status true (or until interrupted).
 - b. Read the Data register to get data and clear interrupt.

For level one interrupts, the most significant four bits of Status tell the nature of the service request:

Status	Interpretation
0000xxxx	Not used.
0001xxxx	10 msec Periodic System Interrupt (PSI).
0010xxxx	Interrupt from one of the special timers.
0011xxxx	Both PSI and special timer interrupting.
0100xxxx	The Data register contains a byte requested by 68000.
0101xxxx	Not used.
0110xxxx	Not used.
0111xxxx	Power-up reset and self test completed successfully.
1000xxxx	Data contains key address (both SHIFT and CTRL pressed).
1001xxxx	Data contains key address (CTRL only).
1010xxxx	Data contains key address (SHIFT only).
1011xxxx	Data contains key address (no SHIFT or CTRL).
1100xxxx	Data contains knob count (SHIFT and CTRL).
1101xxxx	Data contains knob count (CTRL only).
1110xxxx	Data contains knob count (SHIFT only).
1111xxxx	Data contains knob count (no SHIFT or CTRL).

Knob and Timer Details

The knob count is the number of pulses accumulated since the last knob interrupt. The number counts up to a magnitude of 127 then saturates; it does not wrap around. The value returned is a 2's complement signed byte, which is negative for clockwise turns and positive for counterclockwise. This is probably counter-intuitive.

For special timer interrupts (match on time, delayed interrupt, cyclic interrupt) the Data register is defined:

- bit 7 =1 means a time match interrupt.
- bit 6 =1 means a timed delay interrupt.
- bit 5 =1 means a cyclic timer interrupt.

Any combination of these bits may be set. If bit 5 is set, then bits 4 through 0 indicate how many cycles of the cyclic timer have occurred since the interrupt was requested. This is relevant if the 68000 was running for a long time at a higher interrupt level and so did not service the keyboard. The "missed cycle" counter saturates at 31.

The Keyboard at Power-up and Reset

The 8041 powers up in the following state:

- The auto-repeat is random.
- The knob interrupt rate is random.
- The real time is random.
- All the 8041 interrupts are masked.

The 8041 first does a checksum on its ROM, and examines the language and configuration jumpers. If any of these are wrong or invalid, it will keep on trying until they get right (usually forever). When things do get right, the 8041 interrupts on level one, sending \$7 in the upper half of the Status register and \$8E in the Data register.

When the 8041's reset line is pulled, the following defaults are established:

- All real time clock functions and beeper off.
- Pending interrupts cancelled.
- All interrupts masked.
- All pending knob pulses discarded.
- All saved keystrokes (roll-over) discarded.

Interrupt level one will be asserted for about 20 usec after the reset line goes away.

Pascal Interface to the Keyboard

Module **KBD** (in **INITLIB**) exports several procedures which can be used to send commands or read data from the keyboard, avoiding the messy stuff.

Procedure **BEEP** generates a standard tone, which is the same as you'll get by writing an ASCII "bell" character to the standard file **OUTPUT**. Procedure **BEEPER** lets the caller specify the tone and duration. See the relevant keyboard command discussion (§A3) for interpretation of the parameters.

Procedure **KBDCOMMAND** takes a command byte followed by an integer specifying the number of data bytes "numdata" to be sent, and then five byte arguments. The first numdata parameters will be written to the 8041; pass zeros for the remaining parameters.

Function **READ8041BYTE** reads a single byte of data back from the 8041 using proper protocol. The function expects a keyboard service routine (**KBDISR** in the same module) to process an interrupt and put the data byte into a variable. Thus you can't call **READ8041BYTE** from within an **ISR** -- it will hang, because the keyboard generates level one interrupts.

Chapter 13

The Displays

Introduction

The displays are a much simpler subject than the keyboard, but the 9816, 9826 and 9836 displays are all slightly different. This chapter covers all three displays. References to "the display" should be taken to mean all three versions.

The display is a magnetic-deflection raster device which appears in the 68000's address space as a memory-mapped IO device consisting of dual-port memories and some control registers. The hardware is relatively simple and conceptually quite similar across the family. Alpha characters are generated by a ROM containing the raster bit map for each displayable character. Graphic images are stored as rasterized pixel maps which are scanned and displayed. Alpha and graphic images can be displayed simultaneously.

The CRT control signals are generated by a 6845 chip, which is a member of the 6800 family peripheral devices. The 68000 bus and instruction set allow use of 6800 family devices. System software must correctly set up the control registers of the 6845.

9826 Alpha

Dimensions: 190mm diagonal, with useable viewing area about 130mm x 100mm. Up to 25 lines of 50 characters can be displayed, although Pascal uses the bottom line for a typeahead buffer (see 9836 discussion above).

Character set: Same as the 9836 character set, except that certain characters are displayed in slightly different shapes. The character cell is 8 dots wide by 12 dots high, and character cells are displayed on the CRT adjacent to one another (no extra spacing dots between cells; there are $8 \times 50 = 400$ horizontal Alpha dots). A typical character is represented in a 5 by 7 dot matrix, but more dots may be used for descenders.

Attributes: The 9826 has no character attributes such as blinking or inverse video. Since the graphic and Alpha displays are identically sized and spaced and dots are exclusive-or'ed together, it is possible to "simulate" inverse video by turning on graphics dots "behind" the Alpha screen positions.

The bottom n lines of display can be set up to display in half-bright inverse video. This feature has been used in some applications to visually separate soft key labels from the remainder of the screen.

Enabling: The Alpha display is turned on or off by writing a byte to the CRT controller.

9826 Graphics

Dimensions: Graphic and Alpha dots exactly overlay. The raster is 400 dots wide by 300 dots high. Dot pitch is the same in both vertical and horizontal directions (about 0.3mm per dot).

Normally graphic and Alpha dots are exclusive or'ed together, that is, an Alpha dot will invert a graphic dot and vice-versa. In the "soft key area" at the bottom of the screen, graphic dots are full bright over the half-bright background (inclusive or'ed with Alpha).

Enabling: The graphic display is turned on if its memory is accessed with address line A15 low, and turned off if A15 is high.

9816 Alpha

Dimensions: 240mm diagonal, with a useable area of about 160mm x 120mm. Up to 25 lines of 80 characters can be displayed; Pascal uses the bottom line for a typeahead buffer, see 9836 discussion.

Character set: Same as the 9826 Alpha display. The character cell is 8 dots wide by 12 dots high, but characters are displayed in a 10 by 12 matrix (ie there are 800 horizontal Alpha dots). Usually the 2 extra dots are zero. A typical character is 7 by 8 dots, so there are three dot spaces between character cells.

Attributes: The display has no character attributes such as blinking or inverse video. The bottom n lines can be set up to display in inverse video; this has been used to visually distinguish the soft key area from the rest of the screen. In the soft key area, graphic dots are inverted.

Enabling: The Alpha display is turned on or off by writing a byte to the LSI display controller.

9816 Graphics

Graphics is an optional feature in the 9816, not necessarily present in every mainframe.

Dimensions: The graphics raster is 400 dots wide by 300 dots high. Graphics dot pitch is the same horizontally and vertically (about 0.4mm per dot), with Alpha dots half as wide as graphic dots. Graphic dots are exclusive or'ed with Alpha dots on the display (they invert one another).

Enabling: The graphics display is turned on if its memory is addressed with address line A15 low, and turned off if A15 is high.

Alpha Screen Driver Considerations

The general process by which Alpha characters are displayed on the CRT is as follows. There is a dual-port RAM beginning at \$512000 in the 68000's address space. The 68000 stores in consecutive odd-numbered bytes of this memory characters which it wants displayed on the Alpha screen. The CRT controller circuitry reads these bytes sequentially, starting at an address programmed into the 6845 controller, and extracts from a ROM the pixel patterns to be displayed as the raster sweeps across the screen. Characters from sequential odd-numbered Alpha RAM locations are displayed starting at the upper left-hand corner of the screen, sweeping right until the screen line is full, then dropping to the leftmost position of the next lower screen line.

In the 9836, the even-numbered byte of each word of Alpha RAM controls the highlight attributes (inverse video, blinking etc); the Alpha RAM for the 9836 is really 2k words or 4k bytes in size. In the 9826 and 9816, there is no highlight byte and only the odd bytes are present; the memory is 2k bytes in size. Since a "line" in the 9826 is 50 characters wide, there is a good deal of extra memory in the Alpha RAM on that machine.

A programming note when accessing Alpha RAM. In the 9826 and 9816, a bus error will occur if the 68000 tries to access the even-numbered bytes with a byte-wide operation, since they aren't there. The easiest way to write code which will work properly whether or not the highlight bytes are present (ie will work on any flavor of machine) is to always store both the highlight and data byte together in a word-wide operation. Word accesses to Alpha RAM will always work properly, whether for reads or writes.

Because the Alpha RAM is right on the 68000 bus, operations such as scrolling and window management are easily implemented in software. To scroll a region of the CRT, the 68000 simply shifts the whole region "up" or "down" by copying the data and highlight bytes. This is a very fast operation if done with the move-multiple (MOVEM) instruction.

Controlling Character Attributes

In the 9836 only, Alpha characters can be displayed using any combination of these attributes: inverse video, blinking, underlined, half-bright. The display attributes of a character (which is in Alpha RAM but looks to the 68000 like an odd-numbered byte of its address space) are stored in the immediately preceding even-addressed byte of 68000 memory (Alpha RAM).

The attributes are governed by the low-order 4 bits of the attribute byte as follows:

```
bit 0 = 1 iff inverse video
bit 1 = 1 iff blinking
bit 2 = 1 iff underlined
bit 3 = 1 iff half bright
```

The 6845 CRT Controller

This is an LSI device which does most of the dirty work in generating the visible display you see on the Alpha screen. It sequentially steps through the Alpha RAM, fetches the scan-line pixel maps from the character generation ROM, and generates the basic sweep timing signals.

It is told "what to do" by means of a number of byte-wide registers which can be written into by the 68000 but not read. In the 9826 reading them produces a bus error, since they are not wired to generate the correct data acknowledgement bus signals for reading. In the 9836 no bus error is caused but the data returned is garbage.

Writing to a 6845 register is a two-step process. First you must write, to address \$510001 (5308417 decimal), a byte whose value is an integer between zero and seventeen; this selects which of the eighteen controller registers will to be written into next. Then the byte of data to be sent to the controller is written into address \$510003 (5308419 decimal). See the example program below.

The first 10 registers of the 6845 are initialized at boot time with certain magic numbers which are characteristic of the particular mainframe and whether it is refreshing at 50 or 60 hz. Don't change these values; it is possible to damage the CRT drive circuitry by putting in wrong values.

Some of the remaining registers are potentially useful.

Registers 10 and 11

These registers control the height of the cursor, its blink rate, and whether blinking is enabled. (The character position of the cursor is controlled by registers 14 and 15, see below). The height of the cursor is just the number of horizontal scan lines of a character cell during which the cursor is illuminated. For instance, the 9836 character cells are displayed in a matrix 15 dot rows high; the cursor can be "turned on" at its current character position during the scan of cell dot rows 0 through 14 in order to have a cursor as tall as an entire character cell. (For the 9816, use 0 through 13 instead of 14.)

Bits 0-4 of register 10 select the starting dot row of the cursor; row zero is at the top of the character cell. Bit 5 is set to zero if the desired blink rate is 1/16 of the vertical frame rate (slow blink) or set to one for a blink every 1/32nd of the vertical frame rate (fast blink). Bit 6 is zero if the cursor should not blink at all; one if the cursor should blink fast or slow. Here is the full table of meanings for bits 5 and 6:

6	5	meaning
0	0	non-blinking cursor
0	1	cursor non-display mode
1	0	blink at 1/16th field rate
1	1	blink at 1/32nd field rate

Bits 0-5 of register 11 select the ending row of the cursor. If the ending row number is smaller than the starting row number, no cursor appears at all. However, that is not the approved way to turn off the cursor; instead use the cursor non-display mode of the 6845, set by bits 5 and 6 of register 10.

You might want to try the following program to see its effect on the cursor shape. The program expects a register number in decimal, but the register value in binary. The functions HEX and

BINARY are standard in HP Pascal. The normal cursor setting for the Pascal cursor is register 10=01001100 and register 11=00001101.

```
$sysprog$
program cursorsw (input,output);
var
  regselect [hex('510001')]: char;
  crtreg    [hex('510003')]: char;
  reg,byte: char;
  x: integer;
  instr: string[50];

(* Note: the compiler accesses type 'char' as a byte,
   but type '0..255' as a word; it won't make an
   unsigned subrange. *)

begin
  repeat
    writeln;
    write('What register number (enter in decimal)? ');
    readln(x);
    if (x<10) or (x>15) then writeln('Bad register number')
    else
      begin
        reg := chr(x);
        write('Write what value (enter 8-bit binary pattern)? ');
        readln(instr); byte := chr(binary(instr));
        regselect := reg;
        crtreg := byte;
      end;
  until not true;
end.
```

Registers 12 and 13

The 6845 must be told which byte of the 2k of Alpha RAM is the "first" byte (the one displayed at the upper left corner of the CRT). This address is presented as a 14-bit integer; the lower 8 address bits go to R13, the upper 6 bits to R12. (Of course, R12 expects these as a byte with the two highest bits zero.) The values are stored in a manner completely analogous to the method just shown for cursor size control.

Why should you care about this? Only the lower 11 bits of address are needed to span the 2k bytes of Alpha RAM. The upper three bits of the addresses outputted by the 6845 as it sequentially scans Alpha RAM can be used to get special control capabilities. The programmer can affect the addresses which go out by appropriately setting the scan start address.

In the 9836 design, the three upper bits are designated FLD (bit 11), KEYS (bit 12) and TEXT (bit 13). By definition, when the FLD line is high, characters are being displayed in the "soft key area" near the bottom of the screen, and when FLD is low characters are being displayed in the "text" area. The 9836 uses the KEYS and TEXT bits to turn on/off either or both sections of the screen:

KEYS	TEXT	function
0	0	Neither text nor soft key areas are displayed.
0	1	Text area displayed, soft key area off.
1	0	Soft key area displayed, text off.
1	1	Both areas are displayed.

The KEYS and TEXT signals are "statically" controlled by the choice of start address written to R12 and R13. The FLD signal is also governed by the start address, but dynamically: it may toggle from off to on as the scan address increments through its range, indicating where the soft key area starts.

So, by changing the start address, system software can govern the appearance of the soft key area and text areas, or turn off the Alpha display altogether. Note however that changing the start address also changes which character of Alpha RAM appears at the upper left corner of the screen!

The comments above apply equally to the 9826 and 9816. When using this information, don't forget that an 80 character wide screen requires different setup values than a 50 column screen.

Registers 14 and 15

These registers control the cursor position. R14 is the high byte of the address of the cursor within the Alpha RAM address space, and R15 is the low byte of the address. Note that the 6845 compares the entire 14-bit scan address with the 14-bit cursor address to determine when the raster is over the cursor position.

Graphics Screen Driver Considerations

The Graphics display is a much simpler subject. Again it is simply a dual-port memory accessible both by the 68000 and by the CRT display circuitry.

	9836	9826	9816
Width (mm)	210	130	160
Height (mm)	160	100	120
Dots width	512	400	400
Dots height	390	300	300
Dot spacing (mm)	0.4	0.3	0.4
Start address	\$530000	\$530001	\$530001
Ending address	\$537FFF	\$537FFF	\$537FFF
Size of Graphics mem	32k	16k	16k

As this table indicates, the 9836 has much more Graphic memory than the other family members. 9836 Graphic memory is accessed as word-wide, while the 9826 and 9816 are accessed as odd bytes. A little calculation will show that each machine has slightly more memory than needed for graphic display. The extra bytes can be used as ordinary read/write memory, but remember that access to them is slowed by the dual-port usage of the memory.

The display operation is very simple. Bytes are accessed sequentially, and the bits in a byte are shifted out onto the Graphics raster with the most significant bit to the left. The first byte appears at the upper left hand corner, with its most significant bit being displayed first. The 9836 uses 64 bytes per raster line; the 9826 and 9816 use 50 bytes per line.

When accessing Graphics memory, bit 15 of the address is used to control whether Graphics is displayed or not. This means that Graphics memory is "doubly mapped" -- it will respond to the addresses beginning at \$530001, and also to \$53FFFF. Any access in the \$53Fxxx range will turn OFF the display; any access in the \$530xxx range will turn ON the display.

Pascal Access to the CRT

The Pascal system provides two levels of access to the CRT: through the standard file OUTPUT, and by means of procedures and variables exported from the modules comprising the CRT and keyboard drivers.

File System Operations

The standard file OUTPUT, or any text file opened to the volume named 'CONSOLE', will write to the Alpha CRT. Most characters are displayed, with the mapping from byte value to character image being a function of the keyboard language jumpers discussed in the previous chapter. However, some characters have particular interpretations.

character	effect
1	homes cursor to upper left corner.
3	terminates current call to driver.
7	beeps.
8	moves the cursor left one place if possible.
9	clears from present cursor position to end of line.
10	moves the cursor down one place if possible.
11	clears the screen from present cursor to end.
12	homes cursor and clears screen.
13	moves cursor to left end of line.
28	moves the cursor right one place if possible.
31	moves the cursor up one place if possible.
128-143	cause highlighting on 9836; see comments below.

The choices of these characters are largely historical in nature, and don't make as much sense as most people would wish.

In the Pascal system on the 9836 only, the various character highlighting attributes can be activated by writing to the standard OUTPUT file the characters 129 through 143. Once activated, the Pascal CRT driver continues to apply the selected attributes until they are disabled by writing character 128.

The significance of the values is straightforward; as explained under the subheading "Controlling character attributes", above, there are 16 possible combinations of inverse video, blinking, underscore and half-bright attributes. Each control character is simply 128 plus the selected attribute combination.

```
program flashes (output);
begin
  writeln(chr(129), 'inverted', chr(128));
  writeln(chr(130), 'blinking', chr(128));
  writeln(chr(132), 'underlined', chr(128));
  writeln(chr(136), 'half-bright', chr(128));
  writeln;
  writeln(chr(143), 'deluxe to go with everything', chr(128));
end.
```

Scrolling

Pascal treats the CRT as having 24 lines of 50 or 80 characters. Output is always written to the current cursor position, and then the cursor is advanced one place. After passing the right edge of the screen, the cursor wraps to the next line. After passing the lower right-hand corner, the screen is scrolled up one line, the bottom line is cleared, and the cursor is placed at the left edge of the bottom line.

The screen will also upscroll if a linefeed is written in the last line, and will downscroll if a US character is written in the top line.

It is possible to change the dimensions of the CRT scrolling area. This is done by changing `SCREENHEIGHT` and its counterpart `SYSCOM^.HEIGHT`, and changing `MAXY`. `MAXY` is the number of the line from which scrolling occurs; ie the screen scrolls up whenever a `writeln` is done to line `MAXY`.

Lower-Level Access to the CRT

At this point we begin to discuss features which can be gotten at through the `KBD` module (which is also the CRT driver). These features are of course in no way part of the HP Pascal language; they are just goodies that come with the Pascal system.

Cursor Motion

There are at least three ways to move the cursor around.

- Write to the 6845 CRT controller directly; this is not a good idea, because the Pascal system software will then not know what you did. This technique also does not allow you to find out where the cursor is, since the computer cannot read the contents of the CRT controller registers.
- Use the Access Method requests for `GOTOXY` or `GETXY`, passing the `OUTPUT` file or a textfile opened to `'CONSOLE:'` as the `FIB`. This will work neatly with terminals; the terminal driver can translate the AM calls into escape sequences, and it appears nicely orthogonal within the hierarchical structure of the File System.
- Use the FS level calls `FGOTOXY` and `FGETXY`. These procedures each take three parameters: a file name (actually a `FIB`) and two integers:

```
FGOTOXY(F,0,0) move the cursor
```

```
FGETXY (F,XPOS,YPOS) returns cursor position
```

Note that `FGOTOXY` will not put the cursor out of the limits of the scrolling area, and `FGETXY` will return the actual cursor position as opposed to some invalid location which may have been demanded in a call to `FGOTOXY`.

Interrogating the Dimensions of the CRT

The variables SCREENWIDTH and SCREENHEIGHT give this information.

Writing Directly to Screen Locations

The type CRTWORD is a record whose layout corresponds to the highlight and data bytes in Alpha RAM. Type SCRTYPE is an array of CRTWORDS indexed [0..maxint].

The variable SCREEN points to an SCRTYPE; it is correctly set up by the Pascal system, so that SCREEN^[0] is the CRTWORD at the upper left corner of the display. Therefore,

```
screen^[0].character := 'X'    will write 'X' in the upper left
                               corner of the CRT.

screen^[1].character := 'Y'    will write 'Y' in the next
                               (horizontally adjacent) character.

screen^[n*screenwidth].character := 'Z'
                               will write 'Z' in the leftmost
                               character of the n-th line.
```

Fiddling with the Typeahead Buffer

The typeahead buffer actually stores keypresses in the CRT memory itself, which explains why keys become visible in line 25 when typed. Type KEYBUFFERTYPE could just as well be identical to SCRTYPE. The variable KEYBUFFER points to the last line of the display area. KEYBUFLLENGTH tells how many keystrokes are waiting to be read; it is zero when the buffer is empty. KEYBUFSIZE tells the allowed length of the typeahead buffer, which is a function of the screen width.

If you wished to use the 25th display line for some purpose of your own, you could allocate a substitute keybuffer from the heap (using NEWBYTES from module ASM to get the desired amount of space; get 2 bytes of space for every one character of desired typeahead capacity since each keybuffer entry takes one word!) and point KEYBUFFER to the substitute. Set KEYBUFLLENGTH and KEYBUFSIZE. KEYBUFSIZE has a counterpart which should also be set for the sake of safety: SYSCOM^.KEYBUFFERSIZE. The first cell of the buffer is KEYBUFFER^[0].

However, just moving KEYBUFFER is not sufficient. In SYSCOM^ there is a copy of the pointer to keybuffer called KEYBUFFERADDR. You can set it to the new keybuffer address using either the ORD function or a trick record with one field a pointer and the other an integer. You also have to move the "program status area" at the lower right corner of the display. This is where the "run light" appears, along with debugger display of the current line number. This area is accessed by another array of CRTWORD pointed at by variable PROGSTATEINFO and the counterpart SYSCOM^.PROGSTATEINFOADDR. You must move it just like KEYBUFFER was moved, allocating 8 CRTWORDS worth of space somewhere.

The Debugger also puts information in the lower right corner of the CRT. Unfortunately it doesn't use the PROGSTATEINFO pointer; it uses the copy in SYSCOM^. Moreover, the Debugger makes its own copy of this information at the time it installs itself from INITLIB. In effect this means that if you move PROGSTATEINFO^ and use the Debugger, you're liable to

get some surprises. It's hard to predict exactly what these surprises might be (if they were predictable they wouldn't be surprises?)

Turning the Screens On and Off

The exported variables `ALPHASTATE` and `GRAPHICSTATE` are true when the respective screens are visible.

To change the state of either screen, use the system programming language extension `CALL` on the exported hook procedures `TOGGLEALPHAHOOK` and `TOGGLEGRAPHICSHOOK`:

```
call(togglealphahook);
```

Dumping the Alpha or Graphic Screens

To programmatically force the displays to be dumped to the standard system printer, call the exported hooks `DUMPALPHAHOOK` and `DUMPGRAPHICHOOK`:

```
call(dumpgraphichook);
```

You may also wish to implement your own "dump" keys, sending the output to a different display device such as a non-HP graphics printer. After implementing the procedure which will dump the output, assign its name to the appropriate hook procedure variable. These variables get called automatically when the operator presses `<shift>ALPHA` or `<shift>GRAPHICS`.

Chapter 14

The MISCINFO File

Introduction

Certain characteristics of the Pascal human interface can be altered by an optional data file called MISCINFO. This file is read in after INITLIB is loaded but before TABLE or STARTUP runs.

The system expects to find MISCINFO, if at all, on the boot volume. If this file is found, it consists of one record of type ENVIRON (also defined in module KBD). The contents of an ENVIRON record determine such things as:

- Maximum X and Y dimensions of the CRT.
- Constants used in setting up the 6845 CRT controller chip.
- Address of CRT controller chip and CRT memory buffer.
- Address of keyboard typeahead buffer.
- Address of program state information area (debugger info at lower right corner of typeahead line on CRT).
- Character sequences sent to the console device for some standard operations like moving the cursor up, down, left, right; deleting a character; and the EXECUTE ("ETX") and shift-EXECUTE ("ESC") functions.

A little study of the standard constants used (when no MISCINFO is present) to set up the CRT and keyboard will be enlightening.

If you wish, for instance, to use a terminal as the human interface for your workstation, you can do so. In fact that example is presented in the discussion of the IO subsystem later in this book. If you try to do what the example presents, you'll have to create a MISCINFO file with data describing the characteristics of the terminal you use.

The following program is an example of how to create a MISCINFO file. You might wish to do it more simply, say by declaring a file of ENVIRON and writing out a record with just the values you want.

The Listing

```
program setmiscinfo(input,output);
import sysglobals,kbd,fs;

var
  keyboard: text;

procedure change_environ;
const
  default9826 = crtconststype [64,50,49,10,25,9,25,25,0,11,74,11];
  default9836 = crtconststype[114,80,76,7,26,10,25,25,0,14,76,13];
  newmod80    = crtconststype[103,80,79,2,25,9,25,25,0,11,74,11];
var
  lsyscom: environ;
  lfilename: string[40];
  e: file of environ;
  answer: char;
  instring: string[80];

function getval(int,x,y:integer): integer;
var
  n : integer;
begin
  getval := int;
  instring := strltrim(instring);
  if strlen(instring) = 0 then
    begin
      fgotoxy(input,x,y);
      readln(instring);
    end;
  instring := strltrim(instring);
  if strlen(instring) > 0 then
    begin
      strread(instring,1,n,int);
      strdelete(instring,1,n-1);
    end;
  fgotoxy(input,x,y);
  writeln(int : 1,chr(9));
  getval := int;
end;

procedure setparameters;
var
  i : integer;
begin
with lsyscom do
  begin
    crttype := 0;
    with miscinfo do
      begin
        nobreak:=false;
        stupid :=false;
        slowterm:=false;
        hasxycrt:=true;
        write('Internal CRT ? ');
```

```

repeat
  read(keyboard,answer);
until answer in ['Y','y','N','n'];
writeln(answer);
haslcrt:=answer in ['Y','y'];
                                {for displaying long prompts}

hasclock:=true;
canupscroll:=true;
candownscroll:=true;
end;

with crtctrl do
begin
  rlf:=chr(31);
  ndfs:=chr(28);
  eraseeol:=chr(9);
  eraseeos:=chr(11);
  home:=chr(1);
  escape:=chr(0);
  backspace:=chr(8);
  fillcount:=10;
  clearscreen:=chr(0);
  clearline:=chr(0);
  for i:= 0 to 8 do prefixed[i]:=false;
end;

with crtinfo do
begin
  writeln('A  9826 default 50 character screen');
  writeln('B  9836 default 80 character screen');
  writeln;
  write('What CRT type? ');
  repeat
    read(keyboard,answer);
    if answer in ['a'..'z'] then
      answer := chr(ord(answer)-32);
  until answer in ['A'..'E'];
  writeln(answer);
  case answer of
    'A': begin
          height           := 24;
          width            := 50;
          crtmemaddr       := 5316608;{512000 hex}
          crtcontroladdr   := 5308417;{510001 hex}
          keybufferaddr    := 5319008;{512960 hex}
          progstateinfoaddr := 5319092;{5129B4 hex}
          keybuffersize    := 42;
          crtcon           := default9826;
        end;
    'B': begin
          height           := 24;
          width            := 80;
          crtmemaddr       := 5316608;{512000 hex}
          crtcontroladdr   := 5341185;{518001 hex}
          keybufferaddr    := 5320448;{512F00 hex}
          progstateinfoaddr := 5320592;{512F90 hex}
        end;
  end;
end;

```

```

        keybuffersize      := 72;
        crtcon              := default9836;
    end;
end;

writeln;
writeln('Would you like to edit the addresses or');
write ('the control register constants? ');
repeat
    read(keyboard,answer);
    if answer in ['a'..'z'] then
        answer := chr(ord(answer)-32);
until answer in ['Y','N'];
writeln(answer);
if answer = 'Y' then
    begin
        answer := 'N';
        while answer <> 'Y' do
            begin
                setstrln(instrng,0);
                write (chr(12));
                fgotoxy(input,0,1);
                writeln('height           := ',height:1);
                writeln('width           := ',width:1);
                writeln('crtmemaddr      := ',crtmemaddr:1);
                writeln('crtcontroladdr  := ',crtcontroladdr:1);
                writeln('keybufferaddr   := ',keybufferaddr:1);
                writeln('progstateinfoaddr := ',progstateinfoaddr:1);
                writeln('keybuffersize   := ',keybuffersize:1);

                writeln('crtcon[0]       := ',crtcon[0]:1);
                writeln('crtcon[1]       := ',crtcon[1]:1);
                writeln('crtcon[2]       := ',crtcon[2]:1);
                writeln('crtcon[3]       := ',crtcon[3]:1);
                writeln('crtcon[4]       := ',crtcon[4]:1);
                writeln('crtcon[5]       := ',crtcon[5]:1);
                writeln('crtcon[6]       := ',crtcon[6]:1);
                writeln('crtcon[7]       := ',crtcon[7]:1);
                writeln('crtcon[8]       := ',crtcon[8]:1);
                writeln('crtcon[9]       := ',crtcon[9]:1);
                writeln('crtcon[10]      := ',crtcon[10]:1);
                writeln('crtcon[11]      := ',crtcon[11]:1);

                height           := getval(height,           21,1);
                width            := getval(width,            21,2);
                crtmemaddr       := getval(crtmemaddr,       21,3);
                crtcontroladdr   := getval(crtcontroladdr,   21,4);
                keybufferaddr    := getval(keybufferaddr,    21,5);
                progstateinfoaddr := getval(progstateinfoaddr,21,6);
                keybuffersize    := getval(keybuffersize,    21,7);

                crtcon[0]       := getval(crtcon[0],       21,8);
                crtcon[1]       := getval(crtcon[1],       21,9);
                crtcon[2]       := getval(crtcon[2],       21,10);
                crtcon[3]       := getval(crtcon[3],       21,11);
                crtcon[4]       := getval(crtcon[4],       21,12);
            end;
        end;
    end;
end;

```

```

        crtcon[5]      := getval(crtcon[5],      21,13);
        crtcon[6]      := getval(crtcon[6],      21,14);
        crtcon[7]      := getval(crtcon[7],      21,15);
        crtcon[8]      := getval(crtcon[8],      21,16);
        crtcon[9]      := getval(crtcon[9],      21,17);
        crtcon[10]     := getval(crtcon[10],     21,18);
        crtcon[11]     := getval(crtcon[11],     21,19);
        writeln;
        write ('OK? ');
        repeat
            read(keyboard,answer);
            if answer in ['a'..'z'] then
                answer := chr(ord(answer)-32);
            until answer in ['Y','N'];
            writeln(answer);
        end;
    end;

    down{LF}:=chr(10);   up{US}:=chr(31);   right{FS}:=chr(28);
    flush{ACK}:=chr(6); eof{etx}:=chr(3);   left{BS}:=chr(8);
    stop{DC3} :=chr(19); break{DLE}:=chr(16); badch{?}:=chr(63);
    chardel{BS}:=chr(8); altmode{esc}:=chr(27);
                                linedel{DEL}:=chr(127);
    prefix:=chr(0);   etx:=chr(3);   backspace{BS}:=chr(8);
    cursormask := 0;   spare := 0;
    for i:= 0 to 13 do prefixed[i]:=false;
end;

    end; {with lsyscom^}
end; {setparameters}

begin (*change_environ*)
    setparameters;
    writeln;
    writeln('Enter name for MISCINFO file');
    write (' (default is #3:MISCINFO) ? ');
    setstrlen(lfilename,0);
    readln(lfilename);
    if strlen(lfilename) = 0 then lfilename := '#3:MISCINFO';
    rewrite(e,lfilename);
    e^:=lsyscom;
    put(e);
    close(e,'lock');
end (*change_environ*);

begin
    reset(keyboard,'#2');
    change_environ;
end.

```


Chapter 15

Internal Disc Drives

Floppy Control Board

This chapter discusses the interface to the mini-floppy drive which is built into your 9826A or 9836A. The sub-assemblies which comprise the built in mass storage include:

1. 9130K Double-sided, double-density floppy drive
2. 09826-66561 or 09826-66562 Floppy control board

The 09826-66561 control board was used in early 9826As, and is capable of controlling a single disc drive. As of this writing, the 09826-66562 drive control board is the standard control board for the 9826A and 9836A, and can control two disc drives. Software interface to the boards is the same except that a second drive can be addressed by setting the appropriate bit in the extended command register on the 09826-66562.

Theory of Operation

The 09826-66561 and 09826-66562 floppy control boards are based around an Large Scale Integration (LSI) floppy drive controller chip. The chip is a part of the Western Digital Corporation FD179X family of controller chips or equivalent.

The floppy control board is designed to interface the 9826/36 bus with up to two 5.25 inch double-density, double-sided drives. The drive has 270 Kbytes of capacity when formatted to the HP Logical Information Format (LIF) standard. The actual data transfer rate is 16 Kbytes per second with discs which are formatted to interleave one and default track to track stagger.

The standard disc is formatted (per initialization drivers) with 16 sectors per track, two physical tracks per cylinder, and thirty five physical cylinders. As a result, there are 35 cylinders X 2 tracks/cylinder = 70 physical tracks which are available. However, four of the 70 physical tracks are spares for the initialization driver to use if defective tracks are found during initialization. If no spares are needed during initialization, four additional tracks are written after the last user track. Because of this, only 66 logical tracks are available for the user.

Usually, there are no tracks spared during initialization. In this case, the logical track address is the same as the physical track address. When a defective track is found, it is spared. That is, the physical track is skipped over, and the next available good physical track address is used for the next logical track.

Addressing

The address space of the internal mini-floppy is defined as 440000 hex (4456448 decimal) to 44FFFF hex (45219783 decimal). The space is decoded by the 9826/36 motherboard to a single line called chip select floppy (CSF').

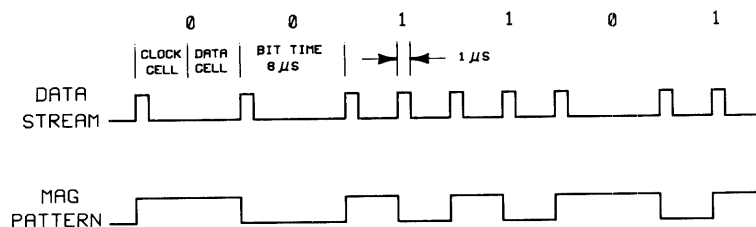
Data transfers

The board has its own internal bus which is used to transfer data to and from the LSI floppy disc controller and the board's 256 byte RAM buffer. Usually the internal bus is connected to the system bus. However, there are some times that the internal bus is "broken off" of the system bus during command execution. During these times the operating system cannot read from or write to any of the memory or registers on the controller board except the extended command and extended status registers.

A state machine on the controller board coordinates data transfers between the floppy disc controller chip and the the 256 byte RAM buffer. This allows the operating system to perform other activities while data is being transferred to or from the disc media. When the controller board sets the interrupt line data transfers are complete.

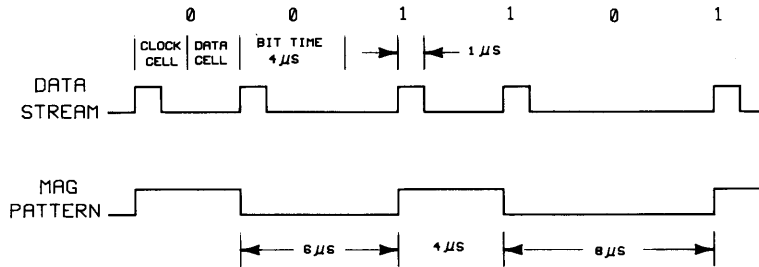
Data Encoding Format

The controller board encodes data onto the disc using the Modified Frequency Modulation (MFM) encoding technique. In Frequency Modulation (FM), data is distinguished to be a one or a zero by the frequency of the data pulses. The system starts with a clock signal of 125 kHz. Clock cell boundaries are defined by a pulse which occurs every 8 μ s. Only clock pulses are present if the bit pattern is all zeros. A 'one' is formed by the adding a pulse to the middle of the 'data' cell. As far as the controller chip is concerned, the addition of the pulse doubles the effective frequency of pulses. In summary, FM encoding has clock cells and data cells which are each 4 μ s long. Every clock cell has a pulse. Only data cells which are ones have pulses.

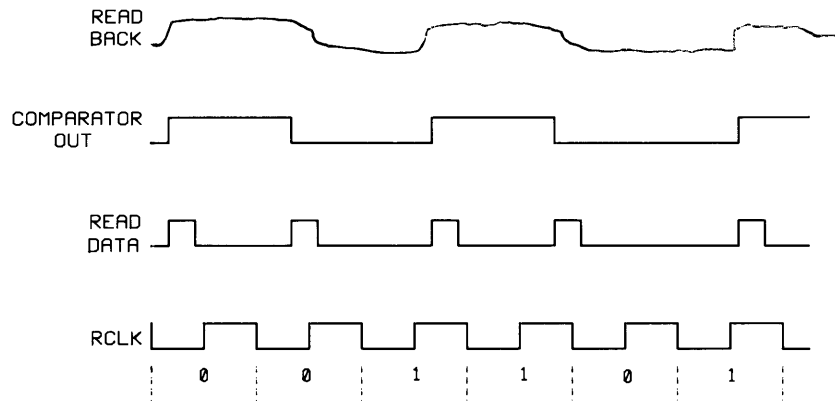


FM DATA ENCODING

MFM encoding is based on a series of clock pulses which occur 4 μ s apart. Only these clock pulses are present if the bit pattern is all zeros. If a one occurs in the bit stream, a DATA pulse is written between the two clock pulses (as in FM encoding) but the CLOCK pulses for the bit and the FOLLOWING bit are NOT written. This MFM encoding technique keeps the transition density low but the data is now phase modulated instead of frequency modulated. That is, identical transition patterns (bit streams) can have different interpretations depending upon the phase of the clock which they start in.



MFM DATA ENCODING



MFM DATA DECODING

Status and Control Registers

A summary of the valid addresses for the controller board is shown below:

09826-66561/66562 ADDRESSES				
REGISTER DESCRIPTION	ADDRESSES		ACCESS	
	HEX	DECIMAL	TYPE	TIME
EXTENDED COMMAND	445000	4476928	WRITE	625 ns
EXTENDED STATUS	445000	4476928	READ	625 ns
CLEAR XSTATUS	445400	4477952	WRITE	625 ns
256 BYTE ON- BOARD RAM	44E000 44E1FF	4513792 4514303	READ-WRITE	1 us
COMMAND	44C000	4505600	WRITE	1 us
STATUS	44C000	4505600	READ	1 us
TRACK	44C002	4505602	READ-WRITE	1 us
SECTOR	44C004	4505604	READ-WRITE	1 us
DATA	44C006	4505606	READ-WRITE	1 us

A detailed description of the controller board registers follows:

Extended Command

The extended command register (at hex address 445000) is an 8-bit write-only register which controls board functions. It is external to the LSI floppy disc controller chip. The bit assignments for the extended command register are as follows:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
ADDR 1	ADDR 0	LOCAL	READ/WRITE	RESET FDC	HEAD 1	PRE-COMP	DRIVE ACTIVE

At power up the extended command (XCMD) register is cleared. This clears bit 3 of the XCMD register which in turn resets the floppy disc controller chip. The XCMD register is a write only register. Because of this, it is up to the mass storage driver to maintain an image of the last command in order to set and clear individual bits. A short description of each bit's function follows:

DRIVE ACTIVE (bit 0) enables the floppy drive which is selected by bits 6 and 7. Setting the DRIVE ACTIVE bit will light the LED on the front panel of the selected drive, and will start and hold the motor on.

PRE-COMP (bit 1) causes a higher level of precompensation current in the read/write head to be used in any writes to the floppy discs. This bit should be set for inner tracks where more pre-compensation current is required (due to the increased bit density) and should be cleared for outer tracks where little or no pre-compensation is needed. Pre-compensation should be used on any track whose number is greater than 16.

HEAD 1 (bit 2) selects the physical head to read and write on the selected drive. This head selection information is NOT latched by the drive. As a result, the state of this bit must remain constant through the duration of a read or write cycle.

RESET FDC' (bit 3) resets the LSI floppy controller chip. The floppy disc controller chip will be reset ANY time this bit is cleared (set to 0). The reset may occur on either the rising or falling clock edge depending upon the implementation of the LSI floppy disc controller chip. This bit is cleared during power up and remains that way until it is set by software.

READ/WRITE' (bit 4) selects the data transfer direction for the state machine which is on the controller board. (A state machine coordinates the transfer of data between the floppy disc controller chip and the board's 256 byte RAM buffer). Setting **READ/WRITE'** to 1 selects a data transfer from the floppy disc controller chip to the 256 byte RAM buffer (**READ** operation). Setting the **READ/WRITE'** to 0 selects a data transfer from the 256 byte RAM to the floppy disc controller chip (**WRITE** operation). The data transfer direction **MUST** agree with the type of disc operation selected, such as read a sector.

LOCAL (bit 5) determines system access to the controller board's internal bus. Setting **LOCAL** to 0 allows system access to the internal bus. Setting **LOCAL** to 1 places the internal bus under the exclusive control of the controller board. The system is not allowed to access the floppy disc controller chip or the 256 byte RAM buffer when **LOCAL** is set. However, **LOCAL** does not affect system access to the extended command or extended status registers. Setting **LOCAL** passes the floppy disc controller chip and 256 byte RAM buffer signals to state machine control. Clearing **LOCAL** gives the responsibility for coordinating data transfer to the system.

ADDRESS 0 and **ADDRESS 1** (bits 6 and 7) are used to select a floppy drive. **ADDRESS 1** is not used in the 9826A and 9836A. Setting **ADDRESS 0** to 0 selects drive 0. Setting **ADDRESS 0** to 1 selects drive 1 (left hand drive in 9836A).

Extended Status

The extended status (**XSTAT**) register (at hex address 445000) is a read-only register which provides information regarding the state of the LSI floppy disc controller chip and the disc drive. The **XSTAT** register is external to the floppy disc controller chip, and provides information which is not available from the floppy disc controller chip. The bit assignments for the extended status register are as follows:

bit 3	bit 2	bit 1	bit 0
INTERRUPT	MARGIN ERROR	MEDIA CHANGE	DATA REQUEST

A short description of each extended status bit follows:

DATA REQUEST (bit 0) is a copy of the data request output of the LSI floppy disc controller chip. This information is primarily used with interrupt information to control the flow of data from the system bus into the floppy disc controller chip during time-critical media initialization operations.

MEDIA CHANGE (bit 1) is set each time the disc drive write protect switch goes from false to true. This will happen every time a disc is fully removed from the drive. The **MEDIA CHANGE** bit is cleared by the clear extended status register (**CLRSTAT**) special function address.

MARGIN ERROR (bit 2) is set when a read data transition occurs too close to a read clock timing signal. This information can be used by during initialization to evaluate media quality. (To aid in determining if a track should be spared). The **MARGIN ERROR** bit is cleared by **CLRSTAT** special function address.

INTERRUPT (bit 3) is a direct copy of the interrupt output of the floppy disc controller chip. Due critical timing needs, the system masks all interrupts during media initialization and uses this copy of the interrupt bit to determine when the LSI floppy disc controller chip has completed its command. This bit is used primarily for fast handshake transfers.

Extended Status Clear

The extended status clear register (at hex address 445400) is a write-only register which, when written to, will clear the extended status register.

Command

The command register (at hex address 44C000) is an 8-bit write-only register which is internal to the floppy disc controller chip. This register holds the command which is presently being executed. With the exception of the 'force interrupt' command, this register should never be loaded with a new command when the floppy controller chip is busy. A summary of the types of valid commands to the floppy controller chip is presented later.

Status

The status register (at hex address 44C000) is an 8-bit read-only register which is internal to the floppy disc controller chip. This register holds floppy drive/command status information such as whether a cyclic redundancy check (CRC) error has occurred during a read command. A summary of the types of status information which is available for each type of command to the floppy controller chip is presented later.

Track

The track register (at hex address 44C002) is an 8-bit read/write register which is internal to the floppy disc controller chip. This register holds the current **READ/WRITE** head position. The track register is incremented by one each time the head is stepped in (towards track 70) and decremented by one each time the head is stepped out (towards track 0) if the update flag is on during **STEP**, **STEP-IN**, and **STEP-OUT** operations. The track register is used during read, write and verify operations. During such operations, the recorded track number in **ID** field from the disc is compared to the contents of the track register. The track register should **NEVER** be loaded when the floppy controller chip is busy.

Sector

The sector register (at hex address 44C004) is an 8-bit read/write register which is internal to the floppy disc controller chip. This register holds the address of the desired sector for read or write operations. During such operations, the recorded sector number in the **ID** field from the disc is compared to the contents of the sector number. The sector register should **NEVER** be loaded when the floppy controller chip is busy.

Data

The data register (at hex address 44C006) is an 8-bit read/write register which is internal to the floppy controller chip. The controller chip internally converts the contents of the data from parallel to serial for write operations, and from serial to parallel during read operations. The floppy controller chip uses the data register to hold the desired track address during seek operations.

On-Board RAM (256 byte buffer)

The floppy controller board contains 256 bytes of local (on-board) RAM. This memory is used during read and write operations to buffer up to a sector (128 words) at a time. When the controller board's internal bus is not in 'LOCAL' mode, the state of each lower address line loaded into an address counter and latched. This address will remain unchanged until local RAM is accessed by the system bus or by the on-board state machine.

When the internal bus is in 'LOCAL' mode, the state machine will increment the address of local RAM after each transfer of data between local RAM and the floppy controller chip. The starting address of these local RAM access is the LAST address read to the system bus. Because of this, it is important that the system reset the local RAM pointer BEFORE passing control to the state machine by reading the first address of local RAM.

The state machine is a shift register and a data selector (address counter). The outputs of the shift register provide appropriate timing and control strobes for read and write operations. The last thing that the state machine does is to increment the local RAM address counter.

Commands and Status

The 9826A/36A internal mass storage controller has eleven basic commands. With the exception of the 'force interrupt' command, the system should never issue a command to the controller board command register while the floppy drive is busy. The floppy controller chip sets a 'DRIVE BUSY' bit (bit 0) in the status register (at address 44C000 hex) while it is executing a command. An interrupt is generated and the 'DRIVE BUSY' status bit reset upon the completion of a command. The status register indicates whether the execution of the last command was fault-free. The eleven basic commands can be divided into four types for ease of discussion as summarized below.

Command Summary

TYPE	COMMAND	BITS							
		7	6	5	4	3	2	1	0
I	Restore	0	0	0	0	H	V	R1	R0
I	Seek	0	0	0	1	H	V	R1	R0
I	Step	0	0	1	U	H	V	R1	R0
I	Step In	0	1	0	U	H	V	R1	R0
I	Step Out	0	1	1	U	H	V	R1	R0
II	Read Sector	1	0	0	M	S	E	C	0
II	Write Sector	1	0	1	M	S	E	C	A
III	Read Address	1	1	0	0	0	E	0	0
III	Read Track	1	1	1	0	0	E	0	0
III	Write Track	1	1	1	1	0	E	0	0
IV	Force Interrupt	1	1	0	1	I3	I2	I1	I0

A description of the command flags, commands and status information for each of the different types of commands follows below. The command flags information is presented first to familiarize you with the option parameters which are available for each command. A description of the actual commands and status information follows the flags section.

Type I Command Flags

- **U** = Update the track register flag. This flag is valid only on the step, step-in and step-out commands. The restore and seek commands cause an automatic update of the track register. Setting this bit to 1 on valid commands causes the track register to be updated, otherwise the track register is left alone.
- **H** = Head load select flag. This flag is valid for all Type I commands. Setting this bit to 1 causes the READ/WRITE head to load at the beginning of the command. Setting this bit to 0 causes the head to be unloaded at the beginning of the command. The head will automatically be disengaged if the floppy controller chip is left idle for more than 15 revolutions of the disc media. **HOWEVER**, the head load solenoid is not implemented in the 9826/36 internal disc drive -- the read/write head is always in contact with the media.
- **V** = Verify flag. This flag is valid for all Type I commands. Setting bit to 1 causes a verification operation to be performed on the destination track. If the verify flag is set to 0 then no verification takes place on the destination track. The floppy controller chip verifies the track by comparing the contents of the track address register to the track address in

the first ID field it encounters. A successful verification occurs when there is a match and the ID field CRC is valid. When this happens, the 'INTERRUPT' bit is set and the 'DRIVE BUSY' bit is reset.

The 'INTERRUPT' and 'SEEK ERROR STATUS' bits are set and the 'DRIVE BUSY' bit reset if no match is found but there is a valid ID field CRC.

The 'CRC ERROR' status bit is set if there is a match but there is no valid ID field CRC. When this happens, the next encountered ID field is read from the disc for verification. The 'DRIVE BUSY' bit is reset and 'INTERRUPT' is set if an ID field with a valid CRC is found within four revolutions of the disc.

- **R1, R0** = Stepping Motor Rate flags. These flags specify the amount of time which each track positioning step is to take. An additional 15 ms is required in the last step if the 'V' (verify) flag is set for any Type I command or if the 'E' flag is set for any Type II or Type III commands. Except for the RESTORE command, R1 and R0 are normally set to 0 in 9826A/36A systems. The stepping rates for each combination of R1 and R0 with the 'E' and 'V' flags cleared are shown in Table X.X.

TABLE X.X STEPPING RATES

R1	R0	milli-seconds
0	0	3
0	1	6
1	0	10
1	1	15

Type II Command Flags

- **M** = Multiple Records flag. This flag is available on all Type II commands. Setting this flag to 0 causes a single sector to be read or written, and setting this flag to 1 causes a multiple sectors to be read or written with the sector register internally updated so that verification can occur when the next record is encountered. Additional sectors will continue to be read until the sector register exceeds the number of sectors on the track or until a 'FORCE INTERRUPT' command is issued to the controller chip. The 'RECORD-NOT-FOUND' bit will be set if the value in the sector register exceeds the number of records on the track. The 'FORCE INTERRUPT' command causes the floppy controller chip aborts the current command and issues an interrupt. **THE FAST HANDSHAKE METHOD OF DATA TRANSFER MUST BE USED TO COORDINATE DATA TRANSFER WHEN THIS COMMAND IS PERFORMED.**
- **S** = Side Select compare flag. This flag is available on all Type II commands and is used for comparison only. (See Side Comparison Flag below). Setting the "S" flag to 0 indicates that side 0 is desired. Likewise, setting it to 1 indicates that side 1 is desired.
- **C** = Side Compare flag. This flag is available on all Type II commands. No side comparison is made if 'C' is set to 0. The side number is read off of the track ID field from the disc and compared to the side select flag 'S'. If a match exists, the floppy controller chip continues with the instruction. The 'INTERRUPT' and 'RECORD-NOT-FOUND' status bits are reset and 'DRIVE BUSY' reset if a match is not found within five revolutions.

- **E** = Delay flag. This flag is available on all Type II commands. Read/write heads are sampled 15 ms after they are loaded if 'E' is set to 1. Otherwise, the read/write heads are sampled immediately after they are loaded.
- **A** = Data Address Mark. This flag is available only on the 'WRITE SECTOR' command. Setting this flag to 1 causes the 'WRITE SECTOR' command to write a 'Deleted Data Mark' on the disc, while setting this flag to 0 causes the write sector command to write a 'Data Mark' on the disc.

Type III Command Flags

- **E** = Delay flag. This flag is available on all Type III commands. Read/write heads are sampled 15 ms after they are loaded if 'E' is set to 1. Otherwise, the read/write heads are sampled immediately after they are loaded.

Type IV Command Flags

- **I3** = Immediate Interrupt. (Requires reset as described below).
- **I2** = Interrupt on Every Index Pulse.
- **I1** = Interrupt on Ready-to-Not-Ready Transition.
- **I0** = Interrupt on Not-Ready-to-Ready Transition.

Note

If interrupt flag bits I3 thru I0 are set to 0, there is no interrupt generated. However, the current command is aborted and the 'DRIVE BUSY' status bit is reset. Issuing the 'FORCE INTERRUPT' command with all interrupt flags set to 0 is the only command which will cause the 'IMMEDIATE INTERRUPT' command to clear.

A description of the basic commands and status information follows immediately below. A detailed description of the command flags for the different types of commands follows this section.

Type I Commands

Restore

This command causes the drive read/write head to seek toward track 0 until the track 0 switch is enabled. The stepping rate is determined by the stepping rate flags R1 and R0. The track register is loaded with zeroes and the 'INTERRUPT' bit set and 'DRIVE BUSY' reset when the command is completed successfully. The 'INTERRUPT' and 'SEEK ERROR' bits are set and 'DRIVE BUSY' reset if the track 0 switch does not enable in less than 255 stepping pulses.

Seek

This command seeks the read/write head from a current track to a desired track. The track register is assumed to contain the current track number. The data register is assumed to contain the desired track number. The floppy controller chip will step the heads in the appropriate direction and update the track register until the contents of the track register match the contents of the data register. 'INTERRUPT' is set and 'DRIVE BUSY' reset at the completion of the command.

Step

This command seeks the read/write heads on step in the same direction as the previous STEP command. The track register is updated (incremented or decremented) if the 'U' flag is on. A verification takes place after a delay determined by the R1 and R0 bits if the 'V' flag is on. 'INTERRUPT' is set and 'DRIVE BUSY' reset at the completion of the command.

Step-in

This command seeks the read/write heads one step in the direction towards track 70. The track register is updated (incremented or decremented) if the 'U' flag is on. A verification takes place after a delay determined by the R1 and R0 bits if the 'V' flag is on. 'INTERRUPT' is set and 'DRIVE BUSY' reset at the completion of the command.

Step-out

This command seeks the read/write heads one step in the direction towards track 0. The track register is updated (incremented or decremented) if the 'U' flag is on. A verification takes place after a delay determined by the R1 and R0 bits if the 'V' flag is on. 'INTERRUPT' is set and 'DRIVE BUSY' reset at the completion of the command.

Type II Commands

Read Sector

This command causes the floppy controller chip to read a sector of data from the disc. The system must seek the read/write heads to the desired track and then load the desired sector number into the sector register prior to issuing the 'READ SECTOR' command. (The side select and compare flags must be set appropriately for this command.) Data is transferred from the disc only after an ID field has been encountered that has the following conditions true:

1. Correct track number
2. Correct sector number
3. Correct side number
4. Correct CRC

The command is aborted and the 'RECORD-NOT-FOUND' status bit set if the 'Data Address Mark' is not found within 43 bytes of the last ID field CRC byte. The 'DATA REQUEST' status bit is set each time a byte is read from the disc. The 'LOST DATA' status bit is set if the system has not read the previous contents of the data register before the data register receives a new data byte from the disc. Passing control to the on-board state machine through the 'LOCAL' bit in the extended status register causes the state machine to coordinate this data transfer.

This sequence continues until the complete sector data field has been transferred from the disc. The 'CRC ERROR' status bit is set and the command aborted if there is a CRC error at the end of the data field. This will happen even if the multiple records flag 'M' is set.

The type of 'Data Address Mark' encountered in the data field is recorded in the status register upon completion of the read operation as follows: 1 = 'Deleted Data Mark' and 0 = 'Data Mark.'

Write Sector

This command causes the floppy controller chip to write a sector of data to the disc. The system must seek the read/write heads to the desired track and then load the desired sector number into the sector register prior to issuing the 'WRITE SECTOR' command. (The side select and compare flags must be set appropriately for this command.) The 'Data Request' status bit is set only after an ID field has been encountered that has the following conditions true:

1. Correct track number
2. Correct sector number
3. Correct side number
4. Correct CRC

The 'Data Request' must be serviced (data register loaded) when the 'Data Request' bit comes true. Otherwise, the command is aborted and the 'LOST DATA' status bit is set. The floppy controller chip writes 12 bytes of zeros followed by the 'Data Address Mark' onto the disc.

It then writes the data field and issues 'Data Requests' to the computer. The 'LOST DATA' status bit is set if the 'Data Requests' are not serviced in time for continuous writing of a sector of data

to the disc. However, the command is not terminated. The controller chip continues to write to the disc until a sector of data has been transferred. It then calculates the two-byte CRC and writes that onto the disc followed by one byte of logic ones. The command is terminated at this time and the 'INTERRUPT' status bit is set and 'DRIVE BUSY' reset.

Passing control to the on-board state machine through the 'LOCAL' bit in the extended status register causes the state machine to coordinate the data transfer from local RAM to the disc.

Type III Commands

Read Address

This command causes the floppy controller to read the first six byte track ID field it encounters from the disc. The 'Data Request' bit is set for each of the six bytes which are transferred. It is left to the mainframe or on-board state machine to transfer the data from the controller's data register to local RAM. The six bytes which are transferred from the track ID are shown below:

bit 1	bit 2	bit 3	bit 4	bit 5	bit 6
-----	-----	-----	-----	-----	-----
TRACK ADDR	SIDE NUMBER	SECT ADDR	SECT LENGTH	CRC 1	CRC 2
-----	-----	-----	-----	-----	-----

The floppy controller chip checks the CRC of the transferred data stream and sets the 'CRC ERROR' status bit if there is a CRC error. Also, the track address of the ID field is written into the Sector Register. The 'INTERRUPT' status bit is set and 'DRIVE BUSY' reset upon completion of the command.

Read Track

This command causes one full track of data to be read from the disc and transferred to the Data Register. Reading starts with the first encountered index pulse and continues until the next encountered index pulse. The 'Data Request' status bit is set for each byte transferred. All bytes on the disc, including gaps, transferred. No CRC checks are performed. 'INTERRUPT' is set and 'DRIVE BUSY' reset upon completion of the command.

Write Track

This command is used for initializing discs (formatting tracks). Writing starts with the leading edge of the first encountered index pulse and continues until the next encountered index pulse. The 'Data Request' bit is set immediately upon receiving the 'WRITE TRACK' command. However, writing will not start until after the first data byte has been loaded into the Data Register. The command is aborted, the 'INTERRUPT' and 'LOST DATA' status bits set, and the 'DRIVE BUSY' status bit reset if the Data Register is not loaded by the time the first index pulse is encountered. A byte of zeros is substituted for actual data if a data byte is not loaded into the data register when it is needed.

The CRC generator is preset by including the hexadecimal number F5 in the outgoing data stream. The control bytes which are used for initialization are shown below.

INITIALIZATION CONTROL BYTES

DATA REGISTER CONTENTS (HEX)	ACTION
00 thru F4	Write 00 through F4
F5	Write A1*, PRESET CRC GENERATOR
F6	Write C2**
F7	GENERATE TWO CRC BYTES
F8 thru FF	Write FC through FF

* Missing a clock transition between bits 4 and 5

** Missing a clock transition between bits 3 and 4

Type IV Commands

Force Interrupt

This command forces the floppy controller chip to set the 'INTERRUPT' status bit upon detection of the condition specified by the option flags.

Note

If interrupt flag bits I3 thru I0 are set to 0, there is no interrupt generated. However, the current command is aborted and the 'DRIVE BUSY' status bit is reset. Issuing the 'FORCE INTERRUPT' command with all interrupt flags set to 0 is the only command which will cause the 'IMMEDIATE INTERRUPT' command to clear.

Status Information

The Status Register is cleared and the 'DRIVE BUSY' status bit set upon receipt of any command except 'FORCE INTERRUPT'. The 'DRIVE BUSY' status bit is reset but the rest of the status bits remain unchanged if a 'FORCE INTERRUPT' command is received while another command is being executed. The Status Register is updated or, in the case of a previous Type I command, cleared and the 'DRIVE BUSY' bit reset if a 'FORCE INTERRUPT' command is received when the floppy controller chip is idle.

The format of the Status Register is as follows:

Status Register Bit Format

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---
S7	S6	S5	S4	S3	S2	S1	S0

The information returned in the Status Register is a function of the previous command executed. A summary of the status information for each command type follows.

BIT	ALL TYPE I COMMANDS	READ ADDRESS	READ SECTOR
S7	'NOT READY'*	'NOT READY'	'NOT READY'
S6	'PROTECTED'	0	0
S5	'HEAD LOADED'	0	'RECORD TYPE'***
S4	'SEEK ERROR'	'RECORD NOT FOUND'	'RECORD NOT FOUND'
S3	'CRC ERROR'	'CRC ERROR'	'CRC ERROR'
S2	'TRACK 0'	'LOST DATA'	'LOST DATA'
S1	'INDEX'	'DATA REQUEST'	'DATA REQUEST'
S0	'DRIVE BUSY'**	'DRIVE BUSY'	'DRIVE BUSY'

* drive is not ready

** controller is busy

*** 1 = 'Deleted Data Mark', 0 = 'Data Mark'

BIT	READ TRACK	WRITE SECTOR	WRITE TRACK
S7	'NOT READY'	'NOT READY'	'NOT READY'
S6	0	'WRITE PROTECT'	'WRITE PROTECT'
S5	0	'WRITE FAULT'	'WRITE FAULT'
S4	0	'RECORD NOT FOUND'	0
S3	0	'CRC ERROR'	0
S2	'LOST DATA'	'LOST DATA'	'LOST DATA'
S1	'DATA REQUEST'	'DATA REQUEST'	'DATA REQUEST'
S0	'DRIVE BUSY'	'DRIVE BUSY'	'DRIVE BUSY'

Programming Considerations

Following is a list of tips to consider when developing custom drivers for the 9826A/36A internal mass storage.

1. Avoid clearing bit 4 (READ/WRITE') of the extended status register when seeking the read/write heads about the disc. Clearing the READ/WRITE' bit allows the possibility of accidentally destroying data by issuing a valid write command to the disc. It is safer to leave the heads in the read sense until a write command is actually to be executed.
2. Reset the local RAM pointer by reading the first address of local RAM (44E000 hex) prior to turning control of the local bus to the controller board's state machine. When in local mode, the state machine will increment the RAM address after each data transfer between the floppy controller chip and the local RAM. The starting address for this process is the LAST ADDRESS READ TO THE OUTSIDE BUS.
3. The state of the floppy disc controller chip shown in the 'DRIVE BUSY' status bit. The state of the actual drive is shown in the 'NOT READY' status bit. The 'DRIVE BUSY' bit is always set when a command is actually being executed, and is reset (cleared) upon completion of a command. The 'INTERRUPT' bit is also set upon the completion of a command.
4. The RESET FDC' bit (bit 3) of the extended command register is normally set to 1. The floppy disc controller chip will reset any time the the RESET FDC' bit is cleared (set to 0) thus aborting any command in progress. As a result, any command which clears the RESET FDC' bit of the extended command register will be immediately aborted.
5. Allow at least 600 milli-seconds for the disc drive motor to come up to speed prior to attempting any read or write commands.
6. Never attempt to seek the read/write heads beyond physical cylinder 40. Such a seek may permanently damage the heads by causing them to impact the floppy dust cover jacket.

Chapter 16

The Power-Fail Option

Introduction

The 9826 and 9836 may be equipped with an optional battery powered back-up supply, which also contains an uninterruptible real-time clock and some non-volatile CMOS RAM. This section describes the features of this option and how they are accessed.

Features

The Power-Fail option contains an 18 volt, 2 amp-hour nickel-cadmium (NICAD) battery with its associated charging and transfer circuitry, a real-time clock, and CMOS RAM which is battery powered when the AC power is off.

The Power-Fail option is controlled by an 8041A microcomputer which provides some user programmable features. Two 5-volt power supplies are included on the Power-Fail circuit board. One insures that the Power-Fail microcomputer and voltage comparators are operating before the rest of the computer comes up, and the other keeps the CMOS circuitry operating when AC power is off.

The word "battery" is generally used in this discussion to denote the entire Power-Fail "smart peripheral", under the control of its 8041 microcomputer.

Power-Fail Behavior

Once the battery turns on and passes its self-test, it may be thought of as having four states: Power Valid, Power Failed, Last Second, and Switched Off. The 8041 may be programmed to interrupt the 68000 via level 7 (non-maskable interrupt) at each transition among these states, or 68000 interrupts may be suppressed. (Obviously there is no interrupt on the transition to Switched Off!)

- **Power Valid:** This is the normal state, when things are running properly. When power fails, the battery will immediately go to Power Failed state.
- **Power Failed:** In this state, the battery provides protective power to the mainframe for a limited time (default 60 seconds). After a delay which is programmable (default zero seconds) the battery will try to interrupt the mainframe with a power-failed interrupt. If power does not return during the protection period or the NICAD battery is about to die, the battery will go to Last Second state. If power returns and stays up for a specified time (default 1 second) the battery returns to Power Valid state.
- **Last Second:** One second after this state is entered, the battery will go to Switched Off state and shut down the computer. After Last Second is entered, the computer **WILL** be shut down even if power comes back.

- **Switched Off:** Once this happens, if the power is restored the computer will go through its normal power-up sequence as if someone had turned on the main power switch.

Note that in Power Failed state, if power is restored but protection time runs out before the power-back delay is elapsed, the battery will go to Last Second anyway.

There is a fourth timer in the battery which is not programmable. Its purpose is to prevent the power supply from heating up too much while the fan is off. It counts up to 60 seconds when there is a power failure, and if it reaches 60 seconds the computer is shut off. This timer is not cleared when power comes back, but counts back down toward zero at half speed. For instance if power was down for 40 seconds, it would have to be on for 80 seconds before a full minute of protection is again available.

Real-Time Clock

The non-interruptible real-time clock is kept as a combination of three pieces of data: a 32-bit timer which counts in 10 msec increments, a record of the timer value when the clock was set, and the time and date when the clock was set.

To figure out the real time, the battery subtracts the current timer count from the timer value when the clock was set, and adds the difference to the time and date when the clock was set. This is a time-consuming operation which is normally only done when the machine is turned on. For moment-to-moment timing while the computer is on, use the keyboard microcomputer which has a number of timing features.

Non-Volatile RAM

The battery contains 128 bytes of battery-powered CMOS RAM. 16 bytes are used by the battery for its own purposes; 112 are available for user-programmed purposes.

This RAM is accessed by moving it into 8041 memory in 16-byte blocks. Commands are available which enable the 68000 to read or modify a block while it is in the 8041's memory.

No standards have been established for how users may allocate space in this RAM, except that the first 16 byte block is reserved for the real-time clock.

Here is the layout of bytes in the first 16 byte block:

byte	use / meaning
0-2	Will be \$0F, \$A5, \$C2 if the battery has been commanded to set the real time since the CMOS RAM woke up; else garbage. You can use these values to verify that the real time is probably meaningful.
3	Least significant byte of time when clock was set.
4	2nd byte of time when clock was set.
5	Most significant byte of time when clock was set.
6	Least significant byte of day number when clock was set.
7	Most significant byte of day when clock was set.
8-11	Value of 32-bit CMOS counter at time when clock was set.
12-15	Used as temporary cell during computation of real time to honor \$41 command.

Interface to the 68000

The 68000 can send commands to the battery by writing to the byte at address \$458021. Reading a byte from this address yields battery status information.

The 68000 can write data bytes to the battery through address \$458001, or read data from the battery via the same byte address.

The battery status register bits are interpreted as follows:

bit	meaning
0	if = 1, there is data ready to read at \$458001.
1	if = 1, command buffer full; if = 0, battery is ready for a command to be written to \$458021. MUST be zero before a command is sent.
2	if = 1, battery is interrupting 68000 on level 7.
5	if = 1 and bit 2 = 1, this is Last Second interrupt.
6	if = 1 and bit 2 = 1, this is power returning interrupt.
7	if = 1 and bit 2 = 1, this is power fail interrupt.

In general the 68000 communicates with the battery by sending a command to the command register, then sending one or more bytes of data to the data register. If the battery is enabled to interrupt the 68000, level 7 (non-maskable) interrupts will signal the mainframe of changes in battery state. Otherwise the 68000 may ask the battery what's up. See commands \$0x and \$C3 below.

Commands to the Battery

The following commands can be sent to the battery.

- \$01 Tells the battery to turn off power. This command is used to discontinue battery protection in order to conserve the charge. It will turn power off even if there is not a power failure; if there is no power failure, the machine will come back up in about one second.

 - \$10 Tells the battery to stop interrupting on level 7. It takes the battery about 200 microseconds to stop interrupting after this command is received. (The command has been received when bit 1 of the status register goes to zero).

 - \$2x Set the interrupt mask. This command disables the three types of interrupt. The lower four bits of the command are:
 - bit 0 -- must be zero.
 - bit 1 -- If one, power fail interrupt disabled.
If zero, enable condition stays unchanged.
 - bit 2 -- If one, power back interrupt disabled.
If zero, enable condition stays unchanged.
 - bit 3 -- If one, last second interrupt disabled.
If zero, enable condition stays unchanged.

 - \$0x Clear the interrupt mask. Used to enable the three types of interrupt. The lower four bits of the command are:
 - bit 0 -- must be zero.
 - bit 1 -- If one, power fail interrupt enabled.
If zero, enable condition stays unchanged.
 - bit 2 -- If one, power back interrupt enabled.
If zero, enable condition stays unchanged.
 - bit 3 -- If one, last second interrupt enabled.
If zero, enable condition stays unchanged.
- Note that command \$0E will be ignored. Only one or two of these bits may be cleared at a time.

Data is written to and read from the CMOS memory through a 16 byte buffer in the 8041's address space. The following four commands have to do with using the CMOS memory and the buffer.

- \$Fx Tell the battery to send a byte from the CMOS buffer to the 68000. The lower four bits of the command act as a pointer to the byte to be sent. Bit zero of the status register will be 1 when the data is ready.

- \$Bx Used to write to the CMOS buffer. The four lower bits of the command act as a pointer to the byte to be written in the buffer. The command is followed by sending the data. The buffer pointer is retained and decremented when a data byte

is received, so if all 16 bytes of the buffer are to be sent, issue command \$BF followed by 16 data bytes.

- \$7x This command tells the battery to load the CMOS buffer with a 16 byte block of CMOS memory. Bit zero must be a zero. Bits one through three tell what block to load, and must indicate 1 through 7; block zero is used by the Real Time Clock.
- \$6x Tells the battery to write the CMOS buffer into one of the 16-byte blocks of CMOS RAM. Bit zero must be zero. Bits one through three tell what block to write. If block zero is written to, the real time will be lost.

The real time is read and written through the same buffer that is used to read and write CMOS memory. The following three commands are used to read and write the real time.

- \$B7 Tells the battery that the next five bytes of data sent will be the real time. The five data bytes must be sent in this order:

- MSB (most significant byte) of days.
- LSB (least significant byte) of days.
- MSB of time of day.
- Second byte of time of day.
- LSB of time of day.

"Days" is an arbitrary integer. "Time of day" is the number of 10 msec ticks since midnight.

- \$40 Tells the battery to set the time to what is in the buffer.
- \$41 Tells the battery to load the buffer with the real time. Then particular bytes of the real time can be requested by the 68000 using these commands:

- \$F7 MSB of day
- \$F6 LSB of day
- \$F5 MSB of time of day
- \$F4 Second byte of time of day
- \$F3 LSB of time of day

There are three ongoing timers that may be read. These are maintained by the 8041 and are all two bytes long; they are "volatile" in that they go away when the machine shuts down. A single timer buffer in 8041 memory is used by the 68000 to access these timers.

- \$82 Tells the battery to load the timer buffer with the value of the non-programmable 60-second power-supply cooling timer.
- \$90 Load timer buffer with the amount of time that power has been back without leaving Power Fail state.

- #94 Load timer buffer with the length of the most recent power failure since power-up. This timer is set to zero whenever the power fail state is first entered.
- \$EB Send the MSB of the timer buffer to the 68000.
- \$EA Send the LSB of the timer buffer to the 68000.
- \$A7 Set the amount of protection time. Command is followed by two bytes of data (MSB first) indicating the protection time in 10-msec tics. Anything greater than 60 seconds will be treated as 60 seconds.
- \$A5 Set the amount of time power must be gone before giving a level 7 interrupt. Command is followed by two data bytes (MSB first). Time is in 10-msec tics.
- \$A3 Set the amount of time power must be back before leaving the power fail state. Command is followed by two data bytes (MSB first). Time is in 10-msec tics.
- \$DB Tells battery to send power status to 68000. The data bits returned are:
- bit 0 -- If one, power is down.
 - bit 1 -- If one, power fail interrupt delay is up.
 - bit 5 -- If one, the AC is gone.
- \$C3 Tells battery to send a status word to the 68000.
- bit 1 -- If one, power fail interrupt is masked.
 - bit 2 -- If one, power back interrupt is masked.
 - bit 3 -- If one, last second interrupt is masked.
 - bit 4 -- If one, battery is in Last Second state and power is about to go away.
 - bit 6 -- If one, the battery is in power fail state.
- \$C6 Tells battery to send 68000 the self-test status. A value of zero means 8041 thinks battery passed self-test. A value of 2 means it failed.
- \$C7 This command tells the battery to send the amount of the last second that has been used up. It is only valid in Last Second state, and returns time in 10-msec tics.

Pascal Programming Interface

Pascal 2.0 doesn't make much use of the Power-Fail option. However, there is a module called BAT which is in the standard INITLIB and exports some useful routines.

BATINIT sets the standard power fail protection to 60 seconds.

BATCOMMAND takes a command byte, followed by a number telling how many bytes of data to send to the battery, followed by five bytes of data. To send, for instance, a command followed by three bytes, use the call:

```
batcommand (commandbyte,3,data1,data2,data3,0,0)
```

with dummy bytes for the unused data arguments.

Function BATBYTERECIEVED waits until a data byte is available from the battery and then returns it to the caller.

Chapter 17

Object Code Format

Introduction

This section describes the structure of object code files accepted by the linking loader. This is the format generated by the Assembler, Pascal compiler, and Librarian.

Purposes of the Object Code Format

- Allow the linking loader to allocate space for global variables, and to relocate and link references to those variables so the loaded code will run properly.
- Allow the linking loader to allocate space for code segments, and to relocate and link references among the code segments so the loaded code will run properly.
- Provide for management of libraries of code modules which have been compiled independently of any program, and can be bound into a program or the system when needed.
- Provide an efficient notation, so that the operations of linking and loading can be very fast and automatic.

Definitions

- **Module:** A module is the basic unit of compiled or assembled code. The Assembler always generates a single module per invocation. The Pascal compiler generates a module for the main program and one for each Pascal **MODULE** declaration in a compilation. This is true even if the source modules are themselves nested within the main program.
- **Library:** The terms "library" and "code file" are completely equivalent. The output of a compilation or assembly is always a library, even if it contains just a single module. A library consists of a directory describing its contents, and modules of code.
- **Interface text, export text, define source:** These terms also are synonyms. The interface text of a module is essentially just the Pascal declaration of the **IMPORT** and **EXPORT** parts of the original source module.
- **External reference, REF:** A location in the object code which needs to be filled in at link time with the address of code or data which is a component of another module.
- **Definition, DEF:** A location in the object code whose final address is a value which may be used to fill in ("satisfy") a **REF** in some other module. **REFs** and **DEFs** are complementary.

- **Global (variable):** A variable declared in the outer block of a main program, or in the EXPORT or IMPLEMENT part of a module. Global variable space is allocated by the loader. Global variables are addressed at some displacement from where 68000 register A5 points.
- **(Loader) Symbol:** The name given to a DEF'ed value. Symbols may represent the beginning of a contiguous area of memory allocated to a module's globals, or the first instruction of an exported procedure, or the main program, or the location of a structured constant in code space.

Structure of a Library File

Library Directory

A library consists of the library directory, followed by zero or more modules. The library directory is structured like a Pascal 1.0 Workstation disc volume directory. This was chosen as a convenience, there is no special significance to the fact. Here are the relevant declarations.

```

const
  blocksize = fblksize;    {the Loader thinks of things in
                           terms of 512-byte blocks relative
                           to beginning of code file}

  vnlength = 7;           {library name length}
  fnlength = 15;          {length of module name}

type
  volname = string [vnlength];
  filename = string [fnlength];    {name of "file" within library}

  daterec = packed record
    year:    0..100;
    day:     0..31;
    month:   0..12;
  end;

  filekind = (untypedfile,      {directory entry}
             badfile,           {bad blocks}
             codefile,          {executable or linkable}
             textfile,          {UCSD format w/Editor envt}
             asciifile,         {LIF ASCII file format}
             datafile,          {file of <type>}
             sysfile,           {system boot file}
             fkind7, fkind8, fkind9, fkind10,
             fkind11, fkind12, fkind13, fkind14, lastfkind);

  dirrange = 0..mmaxint;        {0..maxlongdir}

  direntry =
    record
      dfirstblk: shortint;      (*module starting block*)
      dlastblk: shortint;       (*block following end*)
    {NOTE: for DIR[0], these refer to the library directory itself}
    case dfkind: filekind of

```

```

    untypedfile:          (*library info in DIR[0]*)
      (dvid: volname;     (*name of library*)
       deovblk: shortint; (*block following library*)
       dnumfiles: dirrange; (*num modules in library*)
       dloadtime: shortint; (*time of last modification*)
       dlastboot: daterec); (*most recent date setting*)
    datafile..lastfkind:
      (dtid: filename;    (*title of module*)
       dlastbyte: 1..flbksize; (*1..256 bytes in last block*)
       daccess: daterec)  (*last modification date*)
end; {direntry}

```

The library directory may be thought of as an array of DIRENTRY, starting at byte zero of block zero of the file. The very first DIRENTRY describes the directory itself, in particular DNUMFILES tells how many modules are in the library. Each subsequent DIRENTRY describes one module by giving its name truncated to 15 characters (DTID), the file-relative number of the first block of the module (DFIRSTFLK), and the number of the next block after the end of the module (DLASTBLK).

Module Directory

The organization of a module is:

```

MODULE DIRECTORY
EXT table          lists referenced external symbols
DEF table          lists symbols defined in this module
DEFINE SOURCE     compiler interface specification
TEXT record       stuff to be loaded
REF table          describes objects to be relocated
TEXT record
REF table
...

```

There is a module directory for each module, telling where to find the components of the module; these are different from the directory of the library, which lists the modules themselves. The order in which the parts of a module occur is not specified; the module's directory tells where to find each part. Any number of TEXT records and REF tables may appear in a module.

The module directory contains the following information. This declaration is a sort of pseudo-Pascal where necessary.


```

moduledirectory = packed record
  date:      daterec;      {date of creation}
  revision:  daterec;      {producer's revision date number}
  producer:  char;        {A = assembler, C = compiler,
                          L = linker, etc.}
  systemid:  byte;        {system version number
                          (hard or soft, etc) }
  notice:    string[80];  {copyright notice}
  directorysize: integer; {size of module directory, in bytes}
  modulesize: integer;    {total size of module, in bytes}

  executable: boolean;    {module has start address}
  relocatablebase,       {number of relocatable bytes required}
  relocatablebase,       {current origin of relocatable code}
  globalbase,            {number of global bytes required}
  globalbase,            {current origin of global area
                          (relative to A5) }

  extblock,              {module relative block of EXT table}
  extsize,               {size of EXT table, in bytes}
  defblock,              {module relative block of DEF table}
  defsize,               {size of DEF table, in bytes}
  sourceblock,          {module relative block where
                          DEFINE SOURCE begins}
  sourcesize,           {size of DEFINE source, in bytes}

  textrecords: integer;  {number of TEXT records}

  {Remainder of directory is made up of variable length objects,
  which must be walked through using pointer arithmetic via the ADDR
  system programming extension, or using tricky variant records.
  Strings begin and end on word (even byte) boundaries. The
  directory itself may cross block boundaries. General value or
  address records (GVR's, see description later) occurring below have
  the short variant offset; the offset itself is the length of the
  GVR to assist in stepping quickly through the list.}

  mname: string[ (variable) ]; {name of module; length=1 byte}

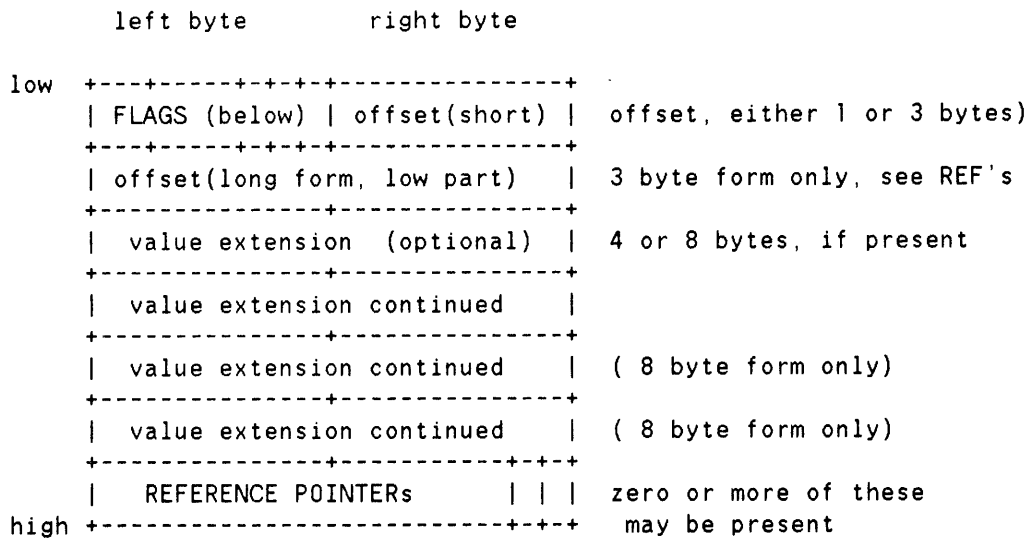
  startaddress:  gvr;        {execution address, present only
                              if executable}

  repeat for each text record {list of TEXT records}
    textstart,   {module relative block of
                  TEXT record}
    textsize,    {size of TEXT record, in bytes;
                  must always be EVEN number! }
    refstart,    {module relative block of
                  REF table}
    refsize:     integer;   {size of REF table, in bytes}
    loadaddress: gvr;       {location to load the TEXT}
  end
end;

```

General Value or Address Record (GVR)

The GVR is a variable length structure which is intended to represent any absolute, relocatable, or linkable value. A GVR begins and ends on a word (even byte) boundary. The format is:



The approximate Pascal type descriptions for a GVR are:

```

type
  reloctype =      (absolute, relocatable, global, general);
  datatype =      (sbyte,          {signed byte}
                  sword,          {signed 16 bit word}
                  sint,           {signed 32 bit integer}
                  fltpt,          {floating point}
                  ubyte,          {unsigned byte}
                  uword);         {unsigned word}

generalvalue = packed record
  primarytype: reloctype; {quick indication of most common types}
  datasize: datatype; {specifies 1, 2, 4 or 8 bytes, signed or not}
  patchable,           {specifies self relative field in branch}
  valueextended: boolean; {indicates presence of valueextension}
  case longoffset: boolean of
    false: (short: 0..255);           {unsigned 8 bits}
    true: (long: 0..16777215);       {unsigned 24 bit value}
end;

valueextension = {present iff valueextended bit set}
  packed record
    case datatype of
      sbyte, sword, sint,
      ubyte, uword: (value: integer);
      fltpt: (valuer: real);
end;

referenceptr = {one or more present if type = general}
  packed record

```

```

    address: 0..16383; {multiply by 4 to get address of EXT symbol}
    op: (addit, subit); {add or subtract the modifying value}
    last: boolean; {indicates end of list}
end;

gvr = concatenation {MOCK PASCAL}
    generalvalue; { 2 to 4 bytes of header info}
    valueextension; { 0, 4 or 8 bytes of value}
    array [zero or more] {list of EXT references}
        of referenceptr;
end;

```

Note

Although the link format provides for REAL arithmetic at link or load time, this feature has not yet been implemented.

The eight-byte form of Value Extension is not used.

Flags

bit 7	6	5	4	3	2	1	bit 0
+-----+-----+-----+-----+-----+-----+-----+-----+							
	primary		data		patch-	value	long
	type		size		able	extend	offset
+-----+-----+-----+-----+-----+-----+-----+-----+							

primary type:

```

00 value type is absolute,      no REFERENCE POINTERS follow
01 value type is relocatable,  no REFERENCE POINTERS follow
10 value type is global,       no REFERENCE POINTERS follow
11 value type is general, with one or more REFERENCE POINTERS

```

data size: (Note that this should be signed long everywhere except in a REF record.)

```

000 signed byte (8 bits)        -128..127
001 signed word (16 bits)       -32768..32767
010 signed long (32 bits)       -2147483648..2147483647
011 floating point (8 bytes)    IEEE 64-bit format
100 unsigned byte (8 bits)      0..255
101 unsigned word (16 bits)     0..65535
110 (reserved)
111 (reserved)

```

patchable:

Indicates that the linker may patch a location in a TEXT record. Used for extending the reach of a short displacement branch by targeting it to a JMP instruction. Applicable only in a REF record, and must be false everywhere else.

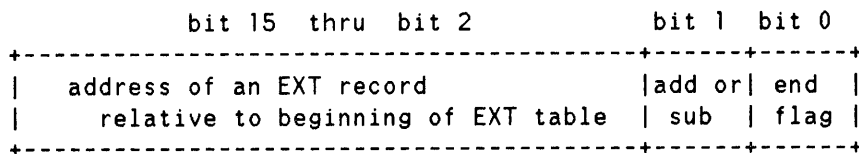
value extend:

- 0 No value extension present, assume 0
- 1 Value extension is present. Length is 4 bytes unless type is floating point, in which case it is 8 bytes. Always true in DEF records.

long offset:

- 0 Use short form (1 byte) of offset field. Value is in the range 0..255 and specifies the total length of the GVR except in REF records. (see DEF record)
- 1 Use long form (3 bytes) of offset field. Value is a 24 bit unsigned number in the range 0..16777215. Applicable only in some REF records.

Reference Pointer



A REFERENCE POINTER is the relative address of an entry in the EXT table. To get the actual address, just clear the end flag (bit 0) and the add/subtract flag (bit 1) and add the rest to the address of the EXT table. Note that this works because 1) EXT records are always a multiple of 4 bytes in length, and 2) the EXT table is limited to 65K bytes in length.

The add or sub flag indicates whether the value of the external symbol is to be 0) added, or 1) subtracted from the GVR value in order to obtain the actual value. There may be any number of REFERENCE pointers in a GVR, and there may be more than one reference to the same EXT record. There may not, however, be both an add reference and a subtract reference to the same symbol, since these would cancel each other.

The end flag indicates whether there are any more REFERENCE POINTERS in this GVR:

- 0 more to come
- 1 this is the last one

There are three special cases for the EXT address:

- **Address 0** (bit pattern 00000000000000xx) refers to the relocation DELTA for the current module (i.e. new load address minus the old load address).
- **Address 4** (bit pattern 00000000000001xx) refers to the global area DELTA for the current module (i.e. new data address minus the old data address).

- **Address 8** (bit pattern 00000000000010xx) is the first valid reference to an external symbol.

There are REFERENCE POINTERS in a GVR only if the primary type field specifies "general". Note that having a primary type of 'relocatable' is an abbreviation for (and entirely equivalent to) having exactly one REFERENCE POINTER with a bit pattern of 0000000000000001, that is, the relocation DELTA should be ADDED to the GVR value exactly once. Similarly, having a primary type of "global" is an abbreviation for (and entirely equivalent to) having exactly one REFERENCE POINTER with a bit pattern of 0000000000000101, that is, the global data area DELTA should be ADDED to the GVR value exactly once.

How a GVR is evaluated

The calculation of a GVR's final value at load time is a process of addition. The effective value is the sum of:

relative part	(relocated base of the segment of code or data)
+ value extension	(if present in the GVR)
+ global part	(if indicated)
+ content of text field	(if and only if the GVR is part of a REF record which is modifying loader text)

This evaluation is performed for the loader by an assembly language routine called EVALGVR. In the Librarian, during linking, it is sometimes necessary to do a more symbolic form of GVR evaluation, where two GVR's are added to yield another GVR rather than an absolute value. This allows modules to be relinked repeatedly. The symbolic evaluation is done by the Librarian's NEWGVR routine.

EXT Table (External Symbol Table)

There may be one EXT table per module. The EXT table begins on a block boundary which is specified in the directory for the module. Its length is also given in the directory. The EXT table is contiguous over its length, which means that individual EXT records within the table may cross block boundaries.

Each EXT record is a packed string which is the name of a symbol referenced (used, imported) in this module, but not defined in it. Each EXT record is a multiple of four bytes long. The first byte of each string is its length (according to the Pascal string type); thus strings may be from 1 to 255 bytes long. If length(string) + 1 is not a multiple of 4, then 1 to 3 bytes are added as padding to make the EXT record extend to the proper boundary. These extra bytes may be garbage!

The first eight bytes of the EXT table are reserved. Thus the first string in the table starts at offset 8 from the start of the table.

The EXT table is restricted to 65532 bytes in length (plus the length of the last string). This is so that any entry in the table can be uniquely referenced by 14 bits. See the description of a REFERENCE POINTER.

```

low  +-----+-----+
      | len = 6 |   S   |
      +-----+-----+
      |   Y   |   M   |
      +-----+-----+
      |   B   |   0   |
      +-----+-----+
      |   L   | padding |
high +-----+-----+

```

EXT record

This one is 8 bytes long.

The formula is:

EXT size = len+4-(len mod 4)

DEF Table (Definition Symbol Table)

There may be one DEF table per module. It contains one DEF record for each symbol which is exported from the module. The DEF table begins on a block boundary which is specified in the directory for the module. Its length is also given in the directory. The DEF table is contiguous over its length, which means that individual DEF records within the table may cross block boundaries.

Each DEF record has two parts. The first part is a packed string containing the name of the symbol which is defined. The string begins and ends on a word (even byte) boundary. If length(string) is even, then an extra byte is added to the end for padding, so that the next part of the DEF record will begin on a word boundary. This extra byte may be garbage.

The second part of a DEF record is a general value or address record (GVR) which defines the value of the symbol which is being exported.

The value extension is 4 or 8 bytes long, according to the datasize field. The value of the symbol is defined to be the value extension plus whatever references are specified by the primary type and any REFERENCE POINTERS that may exist. The value extension must be present.

```

low  +-----+-----+
      | len = 6 |   S   |
      +-----+-----+
      |   Y   |   M   |
      +-----+-----+
      |   B   |   0   |
      +-----+-----+
      |   L   | padding |
      +-----+-----+
      | flags  | len = 8 |
      +-----+-----+
      | value (high part) |
      +-----+-----+
      | value (low part)  |
      +-----+-----+
      | ref pointer  | | |
high +-----+-----+

```

DEF record

(symbol string first)

(GVR second, len is length of GVR part)

(GVR includes any number of REFERENCE POINTERS)

Define Source

There may be one section of **DEFINE SOURCE** per module. It begins on a block boundary, which is given in the module directory. The length is also given in the directory. The **DEFINE SOURCE** may be any arbitrary text, but it is intended to be a copy of the 'define section' from a Pascal separately compiled unit. It is this section of the module which is accessed when it is imported, or used, by the compiler.

TEXT Record

A **TEXT** record is a contiguous chunk of bytes beginning on a block boundary which is given in the module directory. The length is also given in the directory. The **TEXT** record can be any arbitrary data, but is usually object code produced by the compiler or assembler.

A **TEXT** record can be placed by the loader anywhere, as specified by the **GVR** for the load address in the module directory. For example, the FORTRAN 'DATA statement' could be implemented by giving a global data relative value in the load address, which would result in initialization of data when the program is loaded. Another application is the patching of table entries, hooks, or patch jumps, by linking additional modules.

REF Tables

Each **REF** table follows a **TEXT** record and is associated with that **TEXT** record. The **REF** table begins on a block boundary, which is specified in the directory for the module. Its length is also given in the directory. The **REF** table is contiguous over its length, which means that individual **REF** records within the table may cross block boundaries.

Each **REF** record is associated with one object (byte, word, integer or real number) within the preceding **TEXT** record. There can be at most one **REF** record for a given object in the **TEXT** record. The **REF** records are ordered within the table according to the **TEXT** objects they reference.

The **REF** record is basically a general value or address record (**GVR**). (See description elsewhere.)

The offset field specifies which **TEXT** object is referenced. The first **REF** record gives an offset from the beginning of the **TEXT** record. Subsequent **REF** records give an offset from the object referenced by the previous **REF** record. Thus if a **REF** record refers to an object which is less than 256 bytes from the previous one, the short form of the offset may be used. This way, many **REF** records will be only two bytes long!

The data size field refers to the referenced **TEXT** object. The value extension, if it exists, will always be four bytes long and contain that part of the **TEXT** object which can't fit in the **TEXT**. This will only happen for objects of type (8 bit) byte and (16 bit) word which are partially resolved. The true value of the object is the sum of the value in the **TEXT** plus the value extension plus whatever references are specified by the primary type and any **REFERENCE POINTERS** that may exist.

low	+-----+-----+	REF record is a GVR
	flags offset	
	+-----+-----+	offset, 1 or 3 bytes, indicates
	offset (low part)	next object in TEXT record
	+-----+-----+---+	
	ref pointers	(GVR includes any number of
high	+-----+-----+---+	REFERENCE POINTERS)

Miscellaneous Notes

The linker and loader will treat upper and lower case letters as distinct, therefore it is recommended that assemblers and compilers for languages which make no distinction should emit all of their symbols with only uppercase letters.

(This does not apply to module names, in which upper and lower case are equivalent.)

When global data areas are allocated, the base address is at the top of the area, while the base address of a relocatable (code) area is at the bottom of the area.

Chapter 18

The Boot ROMs

Introduction

This chapter of the System Designer's Guide describes the BOOT ROM. User's guide to the BOOT ROM is documented elsewhere (e.g. 9816 Installation Manual, 09816-90000). Also, the tools that are required to build systems are documented elsewhere.

The BOOT ROM is a read only memory located at physical address zero in all of Hewlett-Packard's Motorola 68000 processor based computers. It contains various processor vectors including the power-up vector. It also contains code to load and start up a language or operating system. The code for such a system can be in ROM or stored in some mass storage device. Booting (loading and then granting execution control) of systems is the primary function of the BOOT ROM.

At this time there are at least four versions of the BOOT ROM (1.0 for 9826 only, 2.0 for 9826/9836 only, 3.0 for 9826/9836/9816, and 3.0L for 9816 only). This chapter describes how to use them all. To determine the difference between the 3.0 BOOT ROM and the 3.0L BOOT ROM, look at the power up version number displayed on the screen. The 1.0 and 2.0 BOOT ROMs do not display the version number at power up. (To determine differences from software, see section on BOOT ROM configuration.)

The BOOT ROM also contains evolving code and data segments that may or may not be used by software systems. Most of the code is there to support the booting process itself. This chapter documents the useful code and data segments that can be used safely without fear of change.

Note

Routines in the BOOT ROMs cannot simply be called directly from the Pascal 2.0 system. A small amount of special-purpose interfacing code is required. The section called "Using Boot ROM Routines from Pascal", at the end of this chapter, describes what you must do.

Because the BOOT ROM is physically part of the machine's hardware, the latest BOOT ROM (the 3.0 BOOT ROM) was designed to handle new boot devices and formats that are currently not supported by Hewlett-Packard. Hewlett-Packard may choose not to support other boot devices in the future. The capabilities help assure that a machine with the 3.0 BOOT ROM will be able to accept new I/O cards, new mass storage devices, and new operating systems as they become available.

Note

If you wish to design systems which will be portable to other Hewlett-Packard computers, avoid using the routines in the Boot ROM.

Overview

This chapter is a collection of several documents describing the BOOT ROM.

The BOOT ROM is in some ways the lowest level software kernel of the system. Some of the text that follows is actually more of a system software/hardware interface document, than solely a BOOT ROM document.

Immediately following this section, the most crucial documentation about the BOOT ROM: the boot formats, is presented. The boot formats tell what form that an operating system must be found to be subsequently started by the BOOT ROM.

Setting the default mass storage device specifier (includes: select code, bus address, media format, and device type) is one of the responsibilities of the BOOT ROM. A section of this manual is devoted to how this is done, why it is done, and possible values for the default mass storage device specifier.

Soon, many of Hewlett-Packard's 68000 based products will contain a software readable PROM (programmable read only memory) with the serial number and product number in them. Determining if the PROM is present and reading it are explained.

A whole section of this manual enumerates several recommended techniques for identifying the configuration of the machine. With the aid of these methods, it is possible to design systems that can operate across many of Hewlett-Packard's 68000 based products.

The next section specifies the state of the machine after an operating system has been loaded, but just before the operating system receives execution control.

The next section documents the interfaces which operating systems can use to load additional code and data segments to bootstrap themselves.

A complete set of low level device drivers for the internal 5.25 inch flexible disc drive are also accessible via the BOOT ROM. A section documents them.

The rest of the sections document several miscellaneous things: system switching between operating systems; initializing the internal display; decoding level seven non-maskable interrupts; hanging the machine; a table of character scan-line bit images; high and low memory maps; and boot configurations.

First, the boot formats will be presented.

Boot Formats

This section presents the various forms in which an operating system may exist such that the BOOT ROM will find, load, and start the operating system.

The BOOT ROM will accept two types of systems: hard systems in read only memory (ROM) and soft systems in a wide variety of forms. Hard systems execute primarily out of ROM and are never really loaded (because they are already loaded). Soft systems are always copied (loaded) from some mass storage device into memory and then execute out of that memory.

Explained first, is what the BOOT ROM expects in order to recognize a hard ROM system. Next, the various logical disc media formats and file formats that the BOOT ROM allows for a soft system are enumerated. Then, the ROM/EPROM Disc format allowed for a soft system is explained. Finally, having soft systems on the Shared Resource Manager (SRM) is explained.

Note

Some versions of the BOOT ROM do not support all boot formats. Only the 3.0 and later BOOT ROM versions support everything. In the text that follows, watch for notes indicating limitations.

ROM Headers

A ROM (hard) system can be found in ROM space if it starts with a valid ROM header.

The ROM header is the first 18 bytes found at the beginning of each 16 Kbyte boundary in ROM space. ROM space begins at 256 Kbytes for 1.0 and 2.0 BOOT ROMs, at 64 Kbytes for the 3.0 BOOT ROM, and at 64 Kbytes for the 3.0L BOOT ROM. ROM space ends at 4 Mbytes.

ROM Header (Starting at 16 Kbyte Boundary)	
.....Even Rom.....Odd ROM.....
...Bits 15 thru 8...	...Bits 7 thru 0....
Byte 0	Byte 1
Byte 2	Byte 3
Byte 4	Byte 5
Byte 6	Byte 7
Byte 8	Byte 9
Byte 10	Byte 11
Byte 12	Byte 13
Byte 14	Byte 15
Byte 16	Byte 17

There are two ROM systems that have only 16 bytes of a header (1.0 HPL and 1.0 BASIC). A 16 byte header is present whenever bytes 14 and 15 both have only bit 6 set in bits 7 thru 3. (For all discussions that follow: bit 0 is the least significant bit, LSB.) A 16 byte header is assumed to be part of a pair of 8K by 8 ROMS for checksum purposes.

Here is a description of the ROM header:

Byte	Use
----	---
0	First Byte of Header: Must equal \$F0 for a valid header. (Note "\$" indicates a hexadecimal or base 16 number.)
1	Second Byte of Header: Must equal \$FF to have a valid header.
2	Literal (Third Byte of Header): ASCII character to designate system name. (BOOT ROM only uses this if system ROM bit, bit 0, of byte 3 is set.) Values presently being used are: B for BASIC H for HPL
3	Flag: Flag for type of ROM. Bit 0 1 = System ROM (LSB of the byte) Bit 1 Reserved for HP, set to 0. Bit 2 1 = Language Extension ROM Bit 3 1 = Pseudo-Disc Follows Header in Memory (Not supported by 1.0 or 2.0 BOOT ROMs.) (See ROM/EPROM Disc section.) Bit 4 1 = ROM should not be checksummed Bit 5-7 Reserved for HP, set to 0.
4-7	Reserved: This location can be used for storing a checksum. (How to generate a checksum is shown below.)
8-11	SYSTEM EXECUTE ADDRESS: Address relative to start of header plus 8 to start execution of a system. This is only used if bit 0 of byte 3 is set. It should be zeroed otherwise.
12	EVEN PART ROM Number and SYSTEM REV: Bits 5 thru 0 are the part number for the even ROM in a ROM pair. (1 is the first valid number. 3 is the next valid number.) Bits 7 thru 6 are the system revision number for the even ROM in a ROM pair. (0 is the first valid number.) The 3.0 BOOT ROM verifies that bytes 12 and 13 have consecutive part number values.

- 13 ODD PART ROM Number and SYSTEM REV:
Bits 5 thru 0 are the part number for the odd ROM in a ROM pair. (2 is the first valid number. 4 is the next valid number.) Bits 7 thru 6 are the system revision number for the odd ROM in a ROM pair. (0 is the first valid number.)
- 14 CAPABILITY BITS I and EVEN PART REV:
Bits 2 thru 0 are the part revision for the even ROM in a ROM pair. (1 is the first valid number.) Bits 7 thru 3 are system capability bits:
Bit 7 Reserved for HP, set to 0
Bit 6 A value of 1 means that the system can handle a 50 character wide CRT.
Bit 5-4 Reserved for HP, set to 0
Bit 3 A value of 1 means that the system can handle an 80 character wide CRT
(Capability bits are only looked at by the BOOT ROM if the system bit is set, bit 0 of byte 3.)
- 15 CAPABILITY BITS II and ODD PART REV:
Bits 2 thru 0 are the part revision for the odd ROM in a ROM pair. (1 is the first valid number.) Bits 7 thru 3 are system capability bits:
Bit 7 Reserved for HP, set to 0
Bit 6 Reserved for HP, set to 1
Bit 5-3 Reserved for HP, set to 0
- 16 ROM SIZE and GROUP TYPE:
Bits 7 thru 4 are the ROM size flag. It is a multiple of 64 Kbits (e.g. zero is illegal). (For ROMs that don't have this byte as above, 64 Kbits is assumed.) Bits 3 thru 0 are the group type. Systems are given the number 0. Option ROMs are then given consecutive numbers starting with 1.
- 17 ROM ADDRESS:
This is bits 21 thru 14 of the address of the ROM header byte 0 right shifted 14 bits. The BOOT ROM does not verify this to allow for relocatable systems.

Note

Reserved bits are for future extensions by Hewlett-Packard and could be used by present or future version BOOT ROMs at any time without notice. It is advised to set the reserved bits to the above documented values.

To generate and verify checksums, the following assembly language routine can be used:

```

*
* SUMROM: Checksum a section of memory.
*   On entry:  a0.l = 32-bit start address for checksum
*              d4.l = 32-bit size of area to checksum
*   On exit:   d2.w = 16-bit even (bits 15 thru 8)
*              byte checksum
*              d3.w = 16-bit odd (bits 7 thru 0)
*              byte checksum
*
sumrom  moveq    #0,d1      d1 = Dummy word
        moveq    #0,d2      d2 = Checksum of even bytes
        moveq    #0,d3      d3 = Checksum of odd bytes
sumroml movep.w  0(a0),d5    Get 16 bits from even bytes
        movep.w  1(a0),d6    Get 16 bits from odd bytes
        add      d5,d2      Sum even bytes
        addx     d1,d2      Add in carry bit also
        add      d6,d3      Sum odd bytes
        addx     d1,d3      Add in carry bit also
        addq     #4,a0      Increment to next location
        subq.l   #4,d4      Decrement size to checksum
        bgt.s    sumroml    Repeat if more to checksum
        rts              Done if size is zero or less

```

To generate a checksum, a 32-bit long word location on a four byte boundary inside the area to be checksummed is zeroed. (The address of a 32-bit long word on a four byte boundary has the two least significant bits both equal to zero.) The area must also be a multiple of four bytes in length. Next, the above routine is called with the address of this area in register a0 and the size in bytes of this area in register d4. Next, both odd and even checksums are negated and stored into the previously zeroed checksummed long word. Below is an example of generating a checksum in assembly language:

```

*
* Generating a checksum.
*   On Entry:  addr  equated to Start of Area to
*               checksum on 4 byte boundary
*               size  equated to size of area that
*               is a multiple of 4 bytes
*               chk   equated to address of 32-bit
*               checksum (address must be
*               multiple of 4)
*
        lea      addr,a0      Address of area to checksum
        lea      chk,a1      Address of 32-bit checksum
        move.l   #size,d4    Size of area to checksum
        clr.l    (a1)        Clear Checksum
        bsr     sumrom       Generate checksums
        not     d2           Make even byte compensator
        not     d3           Make odd byte compensator
        movep.w d2,0(a1)     Set even byte checksum
        movep.w d3,1(a1)     Set odd byte checksum

```

To verify a checksum, the sumrom routine can be called again. It should then return -1 for both odd and even byte checksums (registers d2 and d3, respectively).

ROM systems provide the convenience of always being on-line. With a ROM system, the amount of read/write RAM memory required can be minimized.

The current trend of Hewlett-Packard's 68000 based products is towards multiple flavors of languages/operating systems and allowing systems to be improved by revisions. This dictates the use of soft systems for flexibility and re-use of read/write RAM memory.

The next section presents the soft system formats that are recognized by the BOOT ROM.

Boot Disc Formats

When the BOOT ROM searches disc media, it can look for systems on several logical formats. They are: LIF (Logical Interchange Format) 68000 Family System File, SDF (Structured Disc Format) Boot Area, and UNIX ((R) trademark of Bell Laboratories) Boot Area. (All BOOT ROM revisions support LIF. Only the 3.0 BOOT ROM supports all formats.) SDF and UNIX boot areas are supported for possible future use.

To determine which format is on a disc medium, the BOOT ROM looks at the first 16-bit word of the system record. The system record is the first record on a medium (typically 256 bytes in size).

The disc formats are defined below by using Pascal structure type declarations. The following types are used in these definitions:

```

type
  bcd = 0..9;
  bcd12 = packed array[1..12] of bcd;
  unsgn15 = 0..32767;
  string16 = string[16];

```

The actual boot code data is the same for all formats. It consists of a sequence of load segments. Each load segment must begin on a 256-byte boundary (typically the size of a sector). The first 4 bytes is the load address. The second 4 bytes is the number of bytes of code that follow in the load segment. (The load address should be sign extended to the full 32 bits to allow the BOOT ROM to do address range checking before loading.)

Format For A Load Segment

byte #	0	1	2	3	4	5	6	7	8	9	
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----											
	load address				no. of bytes				code text to be		
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----											
	loaded for as many bytes as is required.										

The start address for execution and the total length of the boot code data area is kept in a directory entry or in the header of a boot area depending upon the medium format.

First, the most portable disc format, LIF, is presented.

LIF System File Format

All BOOT ROMs support Logical Interchange Format (LIF). LIF is supported by all 68000 based products as a common means for data communication.

File names must begin with "SYSTEM__" to be found by the BOOT ROM. The 3.0 BOOT ROM also will find system files whose names begin with "SYS".

A LIF disc can have multiple system files on it. The 3.0 BOOT ROM allows any of them to be chosen. (See Installation Guides for more detail, e.g.: 9816 Installation Manual, 09816-90000.) All other BOOT ROM revisions will only allow loading of the first system file found on the disc at power-up. All BOOT ROM revisions allow system switching between multiple systems on a disc. (See section on system switching.)

The LIF system record must be of the following format (concisely shown using Pascal constructs):

```
LIF_vol_type = {LIF volume label}
  packed record
    LIFid:          signed16; {Must equal -32768}
    LIFvolume_label: packed array[1..6] of char;
    LIFdir_start_address: integer;
    LIFoct_10000:   signed16; {Must equal 4096}
    LIFdummy:       signed16; {Must equal 0}
    LIFdir_length:  integer;
    LIFversion:     signed16;
    LIFzero:        signed16; {Must equal 0}
  end;
```

LIFid equal to -32768 identifies the medium to be of LIF format. LIFvolume_label is a six character logical name of the medium. LIFdir_start_address is a 32-bit value pointing to the first sector of the directory. LIFoct_10000 is a 16-bit value that must be equal to 4096 to eliminate console messages on the SYSTEM 3000. LIFdummy is a dummy 16-bit value that must be set to zero. LIFdir_length is a 32-bit value containing the maximum allowable length of the directory in number of sectors. LIFversion is a 16-bit value indicating version of the LIF standard. LIFzero must be set to zero. Words that follow depend upon the version of LIF and are ignored by the BOOT ROM.

A LIF directory entry for a system file to be found by the BOOT ROM must be of the following format:

```
LIF_dir_entry = {LIF directory entry}
  packed record
    LIFfile_name:   pac10 {array[1..10] of char};
    LIFfile_type:   signed16; {Must equal -5822}
    LIFstart_address: integer;
    LIFfile_length: integer;
    LIFtoc:         bcd12;
    LIFl_flag:      boolean;
    LIFvol_number:  unsgn15;
    LIFimplement:   integer; {Execution Address}
  end;
```

LIFfile_name is a 10 character file name. LIFfile_type is a 16-bit number that indicates the file type. -5822 is the file type for a 68000 family product system file and can be found by the

BOOT ROM. LIFstart_address is a 32-bit value that is the starting sector number for the body of the file. (The body of a system file was shown previously.) LIFfile_length is a 32-bit value showing the allocated number of sectors for the file. LIFtoc is 12 BCD digits of the form YYMMDDHHMMSS indicating the time of creation of the file. LIFtoc is ignored by the BOOT ROM. LIFl_flag or last volume flag is one bit. LIFl_flag equal 0 means this is not the last volume of the file. LIFl_flag equal 1 means this is the last volume of the file. LIFl_flag is ignored by the BOOT ROM. LIFvol_number is a 15-bit value indicating the volume number of this file on this medium (zero is illegal). LIFvol_number is ignored by the BOOT ROM. LIFimplement is a 32-bit implementation dependent field. LIFimplement is used by the BOOT ROM as the start execution address in memory for the system.

Below is an example of soft HPL from a LIF 5.25" flexible disc:

Hexadecimal/ASCII dump of record 0 (LIF system record)

```

      +0 +1 +2 +3 +4 +5 +6 +7 01234567
00  80 00 48 50 39 38 32 36  ..HP9826
08  00 00 00 02 10 00 00 00  .....
10  00 00 00 0E 00 01 00 00  .....
18  00 00 00 21 00 00 00 02  .....
20  00 00 00 10 00 00 00 00  .....
28  00 00 00 00 00 00 00 00  .....
30  00 00 00 00 00 00 00 00  .....
38  00 00 00 00 00 00 00 00  .....
40  00 00 00 00 00 00 00 00  .....
48  00 00 00 00 00 00 00 00  .....
50  00 00 00 00 00 00 00 00  .....
58  00 00 00 00 00 00 00 00  .....
.
.
.
F8  00 00 00 00 00 00 00 00  .....

```

LIFid is -32768 or \$8000 hexadecimal.
LIFvolume_label is "HP9826" or \$485039383236 hexadecimal.
LIFdir_start_address is sector 2 or \$00000002 hexadecimal.
LIFoct_10000 is 4096 or \$1000 hexadecimal.
LIFdummy is 0 or \$0000 hexadecimal.
LIFdir_length is 14 or \$0000000E hexadecimal.
LIFversion is 1 or \$0001 hexadecimal.
LIFzero is 0 or \$0000 hexadecimal.
Number of tracks per surface is 33 or \$00000021 hexadecimal.
Number of surfaces per medium is 2 or \$00000002 hexadecimal.
Number of sectors per track is 16 or \$00000010 hexadecimal.

Hexadecimal/ASCII dump of sector 2 (LIF directory entries)

```

    +0 +1 +2 +3 +4 +5 +6 +7 01234567
00 53 59 53 54 45 4D 5F 48 SYSTEM_H First Entry
08 50 4C E9 42 00 00 00 10 PLiB....
10 00 00 01 B1 00 00 00 00 ...l....
18 00 00 80 01 FF FE B7 A0 .... 7
20 63 62 61 63 6B 75 70 20 cbackup Second Entry
28 20 20 E8 14 00 00 01 C1 h....A
30 00 00 00 0B 00 00 00 00 .....
38 00 00 80 01 00 00 05 4B .....K
40 69 62 61 63 6B 75 70 20 ibackup Third Entry
48 20 20 E8 14 00 00 01 CC h....L
50 00 00 00 10 00 00 00 00 .....
58 00 00 80 01 00 00 07 BF .....?
60 39 38 32 35 6B 65 79 20 9825key Fourth Entry
68 20 20 E8 10 00 00 01 DC h....\
70 00 00 00 0B 00 00 00 00 .....
78 00 00 80 01 00 00 05 50 .....P
80 39 38 37 36 63 68 61 72 9876char Fifth Entry
88 73 20 E8 10 00 00 01 E7 s h....g
90 00 00 00 02 00 00 00 00 .....
98 00 00 80 01 00 00 00 9D .....
A0 20 20 20 20 20 20 20 20 End of Directory
A8 20 20 FF FF 20 20 20 20 ..
B0 20 20 20 20 20 20 20 20
B8 20 20 20 20 20 20 20 20
C0 20 20 20 20 20 20 20 20
C8 20 20 20 20 20 20 20 20
D0 20 20 20 20 20 20 20 20
D8 20 20 20 20 20 20 20 20
E0 20 20 20 20 20 20 20 20
E8 20 20 20 20 20 20 20 20
F0 20 20 20 20 20 20 20 20
F8 20 20 20 20 20 20 20 20

```

The first entry (bytes \$00 thru \$19 hexadecimal) is for the 2.0 HPL soft system:

```

LIFfile_name is "SYSTEM_HPL" or $53595354454D5F48504C
hexadecimal.
LIFfile_type is -5822 or $E942 hexadecimal.
LIFstart_address is 16 or $00000010 hexadecimal.
LIFfile_length is 433 or $000001B1 hexadecimal.
LIFtoc is 0 or $000000000000 hexadecimal.
LIFl_flag is 1 or most significant bit of $8001 hexadecimal.
LIFvol_number is 1 or least significant 15 bits of
$8001 hexadecimal.
LIFimplement is $FFFEB7A0 hexadecimal.

```

Next, two media formats that are supported for possible future use are presented.

SDF Boot Area Format

Only the 3.0 BOOT ROM supports Structured Disc Format (SDF).

SDF is a hierarchical directory structure that is used by the Shared Resource Manager (SRM) to manage its files. Some day SDF format may become as common place as LIF is today. It is tree-like, hierarchical, and meets needs of larger file systems. Only the 3.0 BOOT ROM allows the SDF boot area to contain a 68000 system. The purpose of this is to allow the SRM (which runs in a 68000 machine) to boot up straight from one of its own hard discs.

A boot area is a simple block on disc that contains a system to be loaded by the BOOT ROM. There can only be one such area per disc medium. The BOOT ROM cannot decipher the hierarchical directory of SDF directly to find multiple systems (as it can with the LIF format).

The SDF system record contains pointers to the boot area and must be of the following format to be recognized by the 3.0 BOOT ROM (again in Pascal):

```
SDF_vol_type = {SDF volume label}
packed record
  SDFid:          signed16; {Must equal 1792}
  SDFreserved1:  packed array[2..3] of char;
  SDFreserved2:  packed array[1..6] of integer;
  SDFlogical_block_size: integer;
  SDFboot_start_block: integer;
  SDFboot_block_count: integer; {Must be Non-zero}
end;
```

The SDF Boot Block must be of the following format to be recognized by the 3.0 BOOT ROM:

```
SDF_boot_head_type = {SDF boot block header}
packed record
  SDFowner:      signed16; {Must equal -5822}
  SDFexecution_address: integer;
  SDFfilename:   string16;
  {Pad to 256 byte boundary}
  {Load Segments follow as documented previously}
end;
```

SDFowner is 16-bit word that identifies the target machine family for the system. -5822 is the value that identifies the boot area for Hewlett-Packard's 68000 based products. SDFexecution__address is the 32-bit start address where the BOOT ROM starts execution of the system after it has been loaded. SDFfilename is the system name to be displayed in the menu of the 3.0 BOOT ROM. (See 9816 Installation Manual, 09816-90000, for more detail.) The actual load segments then start on the next 256 byte boundary and are of the form previously documented.

SDF is currently used only by the Shared Resource Manager. The next format is supported only for possible future use.

UNIX (R) Boot Area Format

UNIX (R) (registered trademark of Bell Laboratories) Boot Area is supported only on the 3.0 BOOT ROM. This is supported in anticipation of the possibility that someone may want to boot directly from a disc that also contains a UNIX hierarchical file system. (UNIX is an operating system originally designed by Bell Laboratories.)

The first disc sectors of a medium that has a UNIX file system on it has traditionally had system dependent boot information and/or code.

For the 3.0 BOOT ROM to load a system from a UNIX disc, the system record (first sector of the disc) must have the following format:

```
UNIX_vol_type = {UNIX volume label}
packed record
  UNIXid:                signed16; {Must equal 12288}
  UNIXreserved1:        packed array[2..3] of char;
  UNIXowner:            integer; {Must equal -5822}
  UNIXexecution_address: integer;
  UNIXboot_start_sector: integer;
  UNIXboot_byte_count:  integer; {Must be Non-Zero}
  UNIXfilename:         string16;
end;
```

UNIXid is a 16-bit word that indicates that the medium has a UNIX file system on it. UNIXreserved1 is two 8-bit bytes as yet to be defined by Hewlett-Packard. UNIXowner is a 32-bit word that identifies for which machine family the boot area is targeted (-5822 indicates that it is for an HP 68000 based product). UNIXexecution_address is the 32-bit address where the BOOT ROM starts execution of the loaded system. UNIXboot_start_sector is the 256-byte disc sector which starts the contiguous boot area. (Zero is the first sector on the medium. The number of the last sector on the medium depends upon its capacity.) The boot area contains load segments of the same format as described above. The length of the contiguous boot area is given in bytes in the 32 bits of UNIXboot_byte_count. UNIXfilename is the system name displayed in the menu of the 3.0 BOOT ROM. (See 9816 Installation Manual, 09816-90000, for more detail.)

All three of the formats (LIF, SDF, and UNIX) usually implies that they are present on some sort of sector oriented mass storage device. The 3.0 BOOT ROM also allows such formats to be stored in memory in ROM space.

ROM/EPROM Pseudo-Disc Format

To get the best (or worst) of both worlds of hard and soft systems, the 3.0 BOOT ROM can find soft systems stored in ROM/EPROM in the ROM address space (64 Kbytes thru 4M bytes). This allows a soft system to be loaded without the use of any mechanical mass storage device.

A ROM/EPROM pseudo-disc begins with a ROM header in ROM space with the disc bit set (bit 3 of byte 3 in ROM headers section). Immediately following the 18 bytes of header information is the first sector of the disc, the system record. Following the system record are the rest of the disc records. The data is contiguous, except that a zeroed 16-bit word is placed at every 16 Kbyte boundary to prevent ROM headers from accidentally occurring.

A ROM/EPROM pseudo-disc can have any of the previous formats (LIF, SDF, or UNIX).

Unit 0 is the first ROM/EPROM pseudo-disc found in the ROM address space. Unit 1 is the second ROM/EPROM pseudo-disc found in ROM address space. Unit N is the nth ROM/EPROM pseudo-disc found in ROM address space.

All the previous boot format presentations dealt with disc media formats for sector oriented devices that the BOOT ROM read from directly. The 3.0 BOOT ROM also has the capability of talking to Shared Resource Managers (SRM). This is presented next.

SRM System Files

The Shared Resource Manager (SRM) allows multiple machines to share the same operating systems (among other things).

The 3.0 BOOT ROM only, will search the /SYSTEMS directory of each SRM disc volume for 68000 family system files beginning with the name "SYS" for possible load and execution.

For an explanation of how to place and manipulate SRM system files, see FILER chapter of the Pascal 2.0 User's Manual, 98615-90020, December 1982.

That completes the description of the boot formats recognized by the BOOT ROM. Next, a piece of information left by the BOOT ROM for the system, the default mass storage device specifier, is described.

Default Mass Storage

The BOOT ROM is responsible for setting up a variable called the Default Mass Storage Is, DEFAULT_MSUS. It is available for interrogation at any time after a system has been loaded by the BOOT ROM.

DEFAULT_MSUS can be found as a 32-bit value in memory at the hexadecimal address: \$FFFFFFDC.

DEFAULT_MSUS is used by systems for three purposes:

1. Default Mass Storage Device Specifier by language systems (e.g. "msi" statement in HPL or SCRATCH A command in BASIC),
2. Auto-start Device (Device that is searched for a program that is automatically loaded and run at power-up by a language system), and
3. Secondary Load MSUS (place to retrieve additional drivers: e.g. INITLIB of Pascal 2.0).

DEFAULT_MSUS consists of four bytes: the Device TYPE byte, the UNIT Byte, the Select Code Byte, and the Primary Address Byte. The Device TYPE byte specifies the mass storage device and media format. The UNIT byte specifies the drive number, unit number, or volume number. The Select Code byte specifies the I/O select code of the device. The Primary Address byte specifies the HP-IB address or node address. Not all four bytes are used for some device types.

Note

Some device types have already been assigned device numbers even though they are not currently supported and may never be supported. They are assigned because the 3.0 BOOT ROM was designed to provide for the future. Included in this category are: 7905, 7906, 7920, 7925, Bubble memory, and the 5/10/15 Mbyte single volume 5.25" winchesters.

Note

Hewlett-Packard reserves the right to assign any of the reserved device-type numbers at any time.

Below is a breakdown of the assigned values for the four bytes that make up DEFAULT__MSUS.

```
$FFFFFFDC TYPE Byte          This byte defines the File
                              System Format and the
                              Protocol/Device Type.

TYPE<7:5> - Directory Format
  0 = LIF Sector Format
  1 = SDF (Structured Disc Format)
  2 = UNIX Sector Format
  3-6 = Reserved by HP for Future Sector Formats
  7 = Special (ROM, SRM, Networks, etc.)
TYPE<4:0> - Device Type
  0 = 9826/36 internal 5.25" flexible disc
      (or ROM for Special)
  1 = Reserved by HP for Future
      (or SRM for Special)
  2-3 = Reserved by HP for Future
  4 = 9895 8" flexible disc / 913X 5.25"
      winchester (HP-IB)
  5 = 82900 series 5.25" flexible disc (HP-IB)
  6 = 9885 8" flexible disc (GPIO)
  7 = 5-Mbyte 5.25" winchester (HP-IB)
  8 = 10-Mbyte 5.25" winchester (HP-IB)
  9 = 15-Mbyte 5.25" winchester (HP-IB)
 10 = 7905 hard disc (HP-IB)
 11 = 7906 hard disc (HP-IB)
 12 = 7920 hard disc (HP-IB)
 13 = 7925 hard disc (HP-IB)
 14-15 = Reserved by HP for Future
 16 = Command Set '80 devices with
      256-byte blocks (HP-IB)
 17 = all other Command Set '80 devices (HP-IB)
 18-19 = Reserved by HP for Future
 20 = ROM/EPR0M pseudo-disc (ROM Space Memory)
 21 = Reserved by HP for Future
 22 = Bubble memory card)
 23-31 = Reserved by HP for Future
$FFFFFFEDD UNIT Byte          Typically 0 to 3
UNIT<7:0> - Device Dependent Variant Record
            packed record case boolean of
            false: un: 0..255; 8-bit unit number
            true: (vn4: 0..15; vol. no. (CS80/7905/7906)
                   un4: 0..15;) unit number
            end
TYPE<4:0> - Device Type
$FFFFFFEED Select Code Byte  0 to 31
                              (Meaningless for Device type
                              Number 0.)
$FFFFFFEEDF Primary Address  Device Address
Byte                          (HP-IB addr. for HP-IB Discs.)
                              (Node addr. for SRM)
```


The TYPE byte has two fields. Bits 7 thru 5 specify the medium format (e.g. LIF or SDF or Special). Bits 4 thru 0 specify which device it is (e.g. 9895 or 9885). A medium format of type, Special is for boot devices that are communicated with, in a special way (e.g. SRM or ROM).

The UNIT byte specifies the drive number that a medium is found. Most devices such as a 9895 have just one level of specification. For these devices the complete 8 bit value is used. Some devices such as CS-80 discs and the 7096 can have two levels of specification: a volume number and a unit number. This happens for example if there is both a removable and fixed disc platter on the same unit number. In this case the most significant four bits is the volume number and the least significant four bits is the unit number. The UNIT byte may also be meaningless depending upon the TYPE byte. For example a ROM system does not have a valid drive number.

The Select Code byte specifies the I/O card address of the boot device. If the select code is 7, the internal HP-IB is referenced if it is present. The select code byte is device dependent and may also be meaningless depending upon the TYPE byte. (ROM systems do not have a select code.)

The Primary Address byte is a device dependent address field. For some devices it is meaningless (e.g. ROM). It has a different meaning depending upon the device. For an HP-IB disc, it is the HP-IB primary address. For an SRM, it is the node number.

On 1.0 and 2.0 BOOT ROMs, the DEFAULT__MSUS is always set to LIF on an internal flexible disc drive 0.

The DEFAULT__MSUS is set according to the following algorithm:

1. Same as the mass storage that the system was loaded from, for all but ROM systems; or
2. In order of priority, either:
 - a. Non-ROM value passed in DEFAULT__MSUS to booter routine. (The BOOT ROM never does this. Only systems calling the booter can do this.); or
 - b. First device found with media present in boot list (shown below) if ROM is specified in DEFAULT__MSUS. (Media present means that the disc is in, with the door closed and is one of the formats: LIF, SDF, or UNIX); or
 - c. First device found present in the boot list (shown below) if ROM is specified in DEFAULT__MSUS and no media can be found. (A disc drive turned on with the door open fits this category.); or
 - d. A LIF media in an 8290XM drive at HP-IB select code 7, bus address 0, drive 0; if ROM is specified in DEFAULT__MSUS and no devices are present. This was a rather arbitrary choice. INTERNAL,0 was not chosen because case "c" would always catch it. (This case can occur only on machines with no internal mass storage, when a ROM system is powered-up, and no external mass storage devices are powered up.)

When searching for the DEFAULT_MSUS, the 3.0 BOOT ROM uses a priority order called the boot list. All other BOOT ROMs just set the DEFAULT_MSUS to LIF on an internal flexible disc drive 0. The 3.0L BOOT ROM sets the DEFAULT_MSUS to case "d" above. The boot list for the 3.0 BOOT ROM is:

```
Internal flexible disc drive 0
External discs at select codes 0-31, bus address 0,
    unit 0, volume 0
SRM at node 0 at select code 21 on volume 8
Bubble memory card on select code 30
ROM/EEPROM pseudo-disc unit 0
ROM
Internal flexible disc drives 1 thru n
Remaining external discs at select codes 0-31,
    bus addresses 0-7, units 0-7, volumes 0-1
Remaining SRMs at select codes 0-31
Remaining bubble memory cards on
    select codes 0 thru 29 and 31
Remaining ROM/EEPROM pseudo-disc units
```

For each category in the boot list, there is also an order of search. All have some sort of address location. In all cases, lower addresses are found first. This means that select code 0 will be found before select code 7. If a device has multiple addresses to locate it, then searching is done at a local level first. For example, after looking at select code 7, bus address 1, and unit 1; unit 2 at same address will be looked at before going to select code 8. (The SRM equivalent of unit number, the volume, is an exception. It has the strange order of priority: first unit 8, then unit 7, then units 9 thru 24 in order.)

For more information on how the BOOT ROM searches for systems see 9816 Installation Manual, 09816-90000.

CPU Board PROM

In the future, several of Hewlett-Packard's 68000 based products will contain a socketed software readable PROM (programable read only memory) with the serial number and product name in them.

To determine if the PROM is present, look at bit 0 (the least significant bit) of hexadecimal address \$FFFEDA (also known as SYSFLAG2). A value of 1 means that a valid PROM of the format shown below is present. A value of 0 means that there is no PROM present, that the PROM did not checksum, or that the PROM is not Hewlett-Packard format.

Visually one can tell that a machine has a PROM because the 3.0 BOOT ROM will display the serial number on the screen at power up.

The PROM is located at \$5F0001 when present and has data every other byte.

HEX. Address	Bytes Used	Item Description
\$5F0001	002	Checksum
\$5F0005	001	Size of PROM in multiples of 256 bytes
\$5F0007	011	Machine Serial Number represented in ASCII
\$5F001D	007	Product Number represented in ASCII
\$5F004F	078	Used By HP for BOOT ROM Self-Test Information
\$5F00C7	001	Owner Byte: Zero Means it has HP format
\$5F00C9	001	PROM Rev. Byte (Currently zero)
\$5F00CB	002	Spare Checksum (Set to \$FF's) (To allow 2nd Pass to PROM)
\$5F00CF	025	Reserved at \$FF's for HP Future Use
\$5F0101	128	Reserved at \$FF's for Future Use (Half of PROM)

	101	Bytes Allocated
	155	Bytes Reserved for Future Use (Reserved at all \$FF's):

Below is an example of a 9826A with the serial number 2010A000000:

Hexadecimal/ASCII dump of a 9826A PROM

```

      +0 +2 +4 +6 +8 +A +C +E 02468ACE
$5F0001 36 39 01 32 30 31 30 41 69.2010A
$5F0011 30 30 30 30 30 30 39 38 00000098
$5F0021 32 36 41 20 20 FF 01 02 26A ...
$5F0031 03 04 05 FF FF FF FF FF .....
$5F0041 FF FF FF FF FF FF FF FE .....
$5F0051 00 00 FF FF FF FF FF FF .....
$5F0061 FF FF FF FF FF FF FF FF .....
$5F0071 FF FF FF FF FF FF FF FF .....
$5F0081 FF FF FF FF FF FF FF FF .....
$5F0091 FF FF FF FF FF FF FF FF .....
$5F00A1 FF FF FF FF FF FF FF FF .....
$5F00B1 FF FF FF FF FF FF FF 00 .....
$5F00C1 00 00 00 00 00 FF FF FF .....
$5F00D1 FF FF FF FF FF FF FF FF .....
$5F00E1 FF FF FF FF FF FF FF FF .....
$5F00F1 FF FF FF FF FF FF FF FF .....
$5F0101 FF FF FF FF FF FF FF FF .....
$5F0111 FF FF FF FF FF FF FF FF .....
$5F0121 FF FF FF FF FF FF FF FF .....
$5F0131 FF FF FF FF FF FF FF FF .....
$5F0141 FF FF FF FF FF FF FF FF .....
$5F0151 FF FF FF FF FF FF FF FF .....
$5F0161 FF FF FF FF FF FF FF FF .....
$5F0171 FF FF FF FF FF FF FF FF .....
$5F0181 FF FF FF FF FF FF FF FF .....
$5F0191 FF FF FF FF FF FF FF FF .....
$5F01A1 FF FF FF FF FF FF FF FF .....
$5F01B1 FF FF FF FF FF FF FF FF .....
$5F01C1 FF FF FF FF FF FF FF FF .....
$5F01D1 FF FF FF FF FF FF FF FF .....
$5F01E1 FF FF FF FF FF FF FF FF .....
$5F01F1 FF FF FF FF FF FF FF FF .....

```

No two machines will have the same PROM (unless its fraud or non-HP).

The PROM is checksummed to allow for a check of its validity. (See section on ROM headers for checksum algorithm.) A byte is used to determine the size of the PROM to allow for larger size PROMs in the future. The machine serial number is of the form DDDDCSSSSS where DDDD is the date code, C is the country, and SSSSSS is the serial number. The machine serial number is identical to the serial number plate on the back of the machine. The product number (e.g 9826A) is the number marketing gives a machine. It is the first 7 characters or blank pad extended to 7 characters. Together the serial number and product number make up a unique identification for a machine. 78 bytes of the PROM are used by the BOOT ROM to aid in the power-up self-test process. The owner byte specifies who generated the PROM. The revision byte specifies what version the PROM format is. Much of the PROM remains undefined at this time.

The above definition allows for security via the product name and serial number. It allows for reliability via the checksum. It is easy to replace because the serial number matches the one on the back plate. It is easy to trace fraud. (Whenever someone has an illegal PROM serial number,

it will not match their serial number plate. And it will be possible to see whose PROM was copied.)

The serial number PROM is just a small part of machine configuration identification that must be done by operating systems that run on Hewlett-Packard's 68000 family products. The next section presents several recommended techniques for identifying the configuration of the machine. With the aid of these methods, it is possible to design systems that can operate across many of Hewlett-Packard's 68000 based products.

Machine Configuration

Machine configuration identification is important on Hewlett-Packard's 68000 based products if software is to be designed that will operate on more than one product. This section presents the current picture of the evolving set of identification techniques.

The philosophy is to try to determine a capability from the peripheral in question. This means that one shouldn't assume something about a machine based solely upon which product it is. (For example in a 9816, do not assume that there is no CRT alpha underlining capability, instead, look at the SYSFLAG byte to see if CRT alpha highlights are present. Then if the 9816 ever has CRT alpha highlights, the software could use them.)

SYSFLAG

SYSFLAG is a byte at hexadecimal address \$FFFED2 that is part of the current mechanism for communicating machine configuration to operating systems.

SYSFLAG is generated by the BOOT ROM and is defined as follows (bit 0 is the least significant bit):

SYSFLAG<bit 0>	Width of CRT alpha. 0 = CRT alpha is 80 characters wide. 1 = CRT alpha is 50 characters wide.
SYSFLAG<bit 1>	Resolution of CRT graphics. 0 = graphics has 400 horizontal dots by 300 vertical dots. 1 = graphics has 512 horizontal dots by 390 vertical dots.
SYSFLAG<bit 2>	CRT alpha highlights. Highlights include inverse video, underlining, etc. When highlights are not present: (SYSFLAG<2> XOR SYSFLAG<1>) = 0. When highlights are present: (SYSFLAG<2> XOR SYSFLAG<1>) = 1.
SYSFLAG<bit 3>	Internal keyboard controller. 0 = internal keyboard controller present. 1 = internal keyboard controller not present.
SYSFLAG<bit 4>	CRT configuration register (presented below). 0 = no CRT configuration register present. 1 = CRT configuration register present.
SYSFLAG<bit 5>	Internal HP-IB. 0 = internal HP-IB present. 1 = internal HP-IB not present.
SYSFLAG<bit 6>	Reserved for HP, set to a value of 0.
SYSFLAG<bit 7>	Reserved for HP, set to a value of 0.

SYSFLAG is one of many pieces of configuration information available.

SYSFLAG2

There is a second set of system configuration bits in the byte, SYSFLAG2, at hexadecimal address \$FFFFFFEDA. SYSFLAG2 is generated by the BOOT ROM before a system is given control. SYSFLAG2 is defined as follows (bit 0 is the least significant bit):

SYSFLAG2<bits 7:1>	Reserved for HP future use at hexadecimal value of \$27.
SYSFLAG2<bit 0>	Serial number PROM (described earlier). 1 = a valid PROM is present. 0 = no PROM present, that the PROM did not checksum, or that the PROM is not Hewlett-Packard format.

Both SYSFLAG and SYSFLAG2 should be examined a bit at a time (versus examining a whole byte value) to allow reserved bits to take on meanings in the future. Next, the byte, BATTERY, is presented. The whole byte value of BATTERY can be examined.

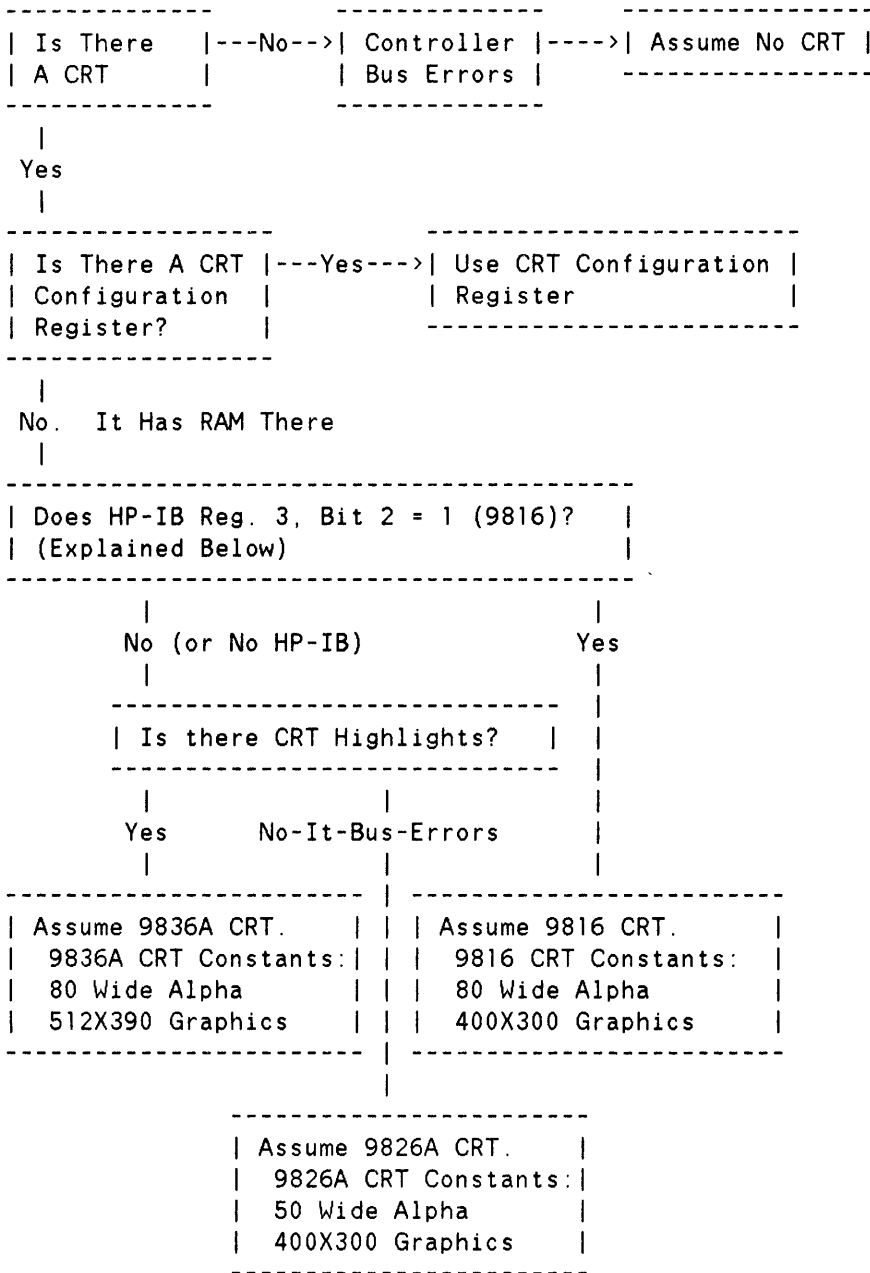
BATTERY

BATTERY is a byte at hexadecimal address \$FFFFFFDCD that contains a value of 1 if the battery backup hardware option is installed and a value of 0 otherwise. BATTERY is initialized by the BOOT ROM at power-up. (The 3.0L BOOT ROM will always set this byte to a value of 0 indicating no battery is present.)

Only a small portion of all configuration information required by operating systems is left by the BOOT ROM. Most information must be extracted by the operating systems themselves. Of this information, some must be inferred as the result of some algorithm. The next section presents how to get additional CRT configuration information.

CRTID, CRT Presence, Graphics Presence

Determining characteristics of the CRT display that are not specifically given by SYSFLAG can be done using the algorithm shown below:



To determine if the CRT is present, a bus error check should be done with the instruction:

```
move.b    #15,$510001
```

(This instruction sets the CRT controller to point to the cursor register.) (See MC68000 User's Manual, 09826-90073, for handling of the bus error exception.)

To determine if the CRT has a CRT configuration register, look at bit 4 of SYSFLAG as explained previously. (The contents of a CRT configuration register are enumerated below.)

CAUTION

If a CRT configuration register is not present, do not read or write the non-existent register. If a non-existent CRT configuration register is accessed, it will turn off the CRT horizontal sweep in some machines (9826 and 9836). Later, accessing of normal CRT alpha will then turn the CRT horizontal sweep back on. This will in turn stress the CRT hardware.

If the machine does not have a CRT configuration register, the CRT must be the same as one of the three CRTs found on the 9826, 9836A, or 9816.

First, a check should be made to see if the CRT is the same as a 9816. This is done by looking to see if bit 2 of the byte at hexadecimal address \$478003 (known as register 3 of the internal HP-IB) is equal to one. Of course the internal HP-IB must be present for this bit to be valid. Bit 5 of SYSFLAG tells if this register and the internal HP-IB are present.

If the machine doesn't have a CRT configuration register and it does not identify as a 9816 like CRT, then a check is made to see which of the two CRT tops it has (9826 or 9836A). This is done by determining if it has alpha highlight capability. A 9836A like CRT has highlight capability and a 9826 like CRT does not.

Below is the 16 bit CRT configuration identification register (at hexadecimal address \$51FFFE) to be used in future products (bit 0 is the least significant bit):

- CRTID<bit 15> - Self-Initializing CRT.
- CRTID<bits 14 thru 13> - Reserved by HP for future, set to 0.
- CRTID<bits 12 thru 11> - Graphics Memory Layout:
(Currently only zero is used.)
0: Monochrome
1: 4 Memory Planes starting at \$520000
4 bits/byte using 256 Kbytes
1 byte/pixel
(least sign. 4 bits)
2: 3 Memory Planes starting at \$528000
8 pixels/byte using 128
Kbytes
(unused 64 Kbytes at \$520000)
3: 8 Memory Planes starting at \$520000
8 bits/byte using 256 Kbytes
1 byte/pixel
- CRTID<bit 10> - Same as SYSFLAG<Bit 2>. CRT alpha highlights. Highlights include inverse video, underlining, etc. When highlights are not present: (SYSFLAG<2> XOR SYSFLAG<1>) = 0. When highlights are present: (SYSFLAG<2> XOR SYSFLAG<1>) = 1.
- CRTID<bit 9> - Same as SYSFLAG<Bit 1>. Resolution of CRT graphics. 0 = graphics has 400 horizontal dots by 300 vertical dots. 1 = graphics has 512 horizontal dots by 390 vertical dots.
- CRTID<bit 8> - Same as SYSFLAG<Bit 0>. Width of CRT alpha. 0 = CRT alpha is 80 characters wide. 1 = CRT alpha is 50 characters wide.
- CRTID<bits 7 thru 4> - CRT Number: Implies CRT constants for BOOT ROM initialization of CRT controller and physical dimensions of CRT.
0: 9826A Monitor
1: 9836A Monitor
2: 9816 Monitor
3-15: Reserved by HP for future products
- CRTID<bit 3> - 1 = 50 Hz. 0 = 60 Hz.
- CRTID<bits 2:0> - Model # (0 is for all systems based on the 6845 CRT controller chip.)

Any CRT top with a CRT ID Register may also have a control register (which is a 16 bit word at hexadecimal address \$51FFFC) to control the CRT:

bit 0 (least significant)	0 = Turns Off Graphics
	1 = Turns On Graphics

Graphics presence is determined by doing a bus error check of graphics RAM.

Besides the CRT, the keyboard has additional configuration information, that is not documented elsewhere.

Keyboard

The keyboard system jumpers are used to identify the configuration of the keyboards. The jumpers can be found on the printed circuit board that has the key switches connected to it. The 9836 and 9826 already use the value of zero. The 9816's small keyboard uses the value of 1. (See other chapters for more detail on the keyboards.)

In addition to knowing which size keyboard is present, one may want to know how many internal flexible disc drives are present.

NDRIVES

The byte NDRIVES at hexadecimal address \$FFFFFFD8 is the highest allowable unit number for the internal flexible disc. A value of 255 (or -1) means that there are no drives present (e.g. 9816). This location is not valid on 1.0 BOOT ROMs. (The BOOT ROM I.D. in the 16 bit word at hexadecimal address \$3FFE is negative for 1.0 BOOT ROMs.)

As the number of BOOT ROMs increases, the need to differentiate between their capabilities arises. The next section presents the BOOT ROM configuration and I.D. words.

BOOT ROM Configuration and Revision

Currently there are two 16 bit words that help to differentiate the ever growing number of BOOT ROMs: the configuration word and the revision word.

The BOOT ROM configuration word is a 16 bit word at hexadecimal address \$3FFC. It is only valid on 3.0 or later revision BOOT ROMs. (To tell which BOOT ROM one has, look at BOOT ROM Revision Word below.) For the 1.0 and 2.0 BOOT ROMs assume the hexadecimal value of \$0501. A layout of the configuration word is shown below (bit 0 is least significant):

BOOT ROM Configuration Word

bits 15 thru 11	Reserved by HP, set to Zero
bit 10	0 = BOOT ROM has user I/O vectors 125 thru 255 defined. (See section on LOW ROM Map exception vectors for more detail.)
bit 9	0 = BOOT ROM has character table at hexadecimal address \$4000 defined. (See section on Character Table for more detail.)
bit 8	0 = BOOT ROM has a Read Interface. (See section appearing later on the Read Interface)
bits 7 thru 5	These bits are reserved by HP, set to zero.
bits 4 thru 0	These bits form a value that indicates the size of BOOT ROM in 16 Kbyte increments.

The BOOT ROM revision word is a 16 bit word at hexadecimal address \$3FFE. It is present in all revisions of the BOOT ROM. Its possible values are:

BOOT ROM Revision Word

Negative (-19492)	1.0 BOOT ROM (9826 only)
1	2.0 BOOT ROM (9836/9826 only)
3	3.0 BOOT ROM

To determine the difference between the 3.0 BOOT ROM and the 3.0L BOOT ROM, look at the BOOT ROM configuration word. The 3.0L BOOT ROM has no read interface, no character table, and does not have user I/O vectors 125-255. (The 3.0 BOOT ROM has everything.) Currently the 1.0, 2.0, and 3.0L BOOT ROMs are 16 Kbytes in size. The 3.0 BOOT ROM is 48 Kbytes in size.

This completes the section devoted solely to machine configuration. The next section summarizes what state the machine is left in by the BOOT ROM just before control is passed to a loaded system.

CPU State at Load

At the time control is turned over to the system, the following conditions will be in effect and the following information will be made available:

1. The type of boot as a 16 bit Value of BOOTTYPE (at hex address \$FFFDC0).

Boot type	Value
POWER UP	0
REQUESTED RE-BOOT	12
REQUESTED BOOT	18

2. Address of lowest usable RAM (determined by RAM test) is stored in LOWRAM at \$FFFFDCE.

3. The flag byte showing battery backup presence (BATTERY) is set up.

4. Interrupts will be disabled:

The hardware will be RESET except for the keyboard, battery backup, and the internal flexible disc. Keyboard interrupts are disabled. There may be something in the keyboard input buffer. Battery backup (if present) is set to give 60 seconds of protection. (60 seconds is the maximum possible.)

5. All vectors above hexadecimal address \$FFFFFFE0 will be initialized to JSR CRASH (jump to subroutine named CRASH). CRASH is a recovery routine in the BOOT ROM. If there is a RAM Monitor present at hexadecimal address \$00880000 (by looking for the hexadecimal value of \$4EF9 at that address), then trap 15 and the trace trap will be left as the monitor set them up. (See MC68000 User's Manual, 09826-90073, for handling explanations of trap instructions.)

6. The status register will be set to a hexadecimal value of \$2700. That is, interrupts will be disabled.

7. The stack pointer will be pointing to the top of the booter stack area (hexadecimal address \$FFFFDAC).

8. SYSFLAG and SYSFLAG2 will be set correctly.

9. The system CRT controller will be initialized to its power-up values. The CRT will be blanked out. Graphics memory will be zeroed and turned off.

10. The DEFAULT__MSUS at \$FFFFFEDC will be set up.

11. NDRIVES will be set up.

12. F__AREA (Low RAM) Memory Pointer is set up. F__AREA is an area of stolen RAM for BOOT ROM usage (mass storage drivers) bounded by the bottom of physical memory and the address stored in LOWRAM at hexadecimal address \$FFFFDCE. BOOT ROM 1.0 steals no space. BOOT ROM 2.0 steals 32 bytes. BOOT ROM 3.0 steals 160 bytes. The 3.0L BOOT ROM steals 44 bytes.

After a system has been loaded and given execution control by the **BOOT ROM**, the above conditions will be valid. If the system that is loaded needs to load additional drivers before it can operate (i.e. mass storage drivers), then the system can do that via the interfaces documented in the next section.

Read Interface and Secondary Loading

There are two methods for operating systems and secondary loaders to utilize the read mass storage drivers of the **BOOT ROM**:

1. The Read Interface drivers which can be redirected to many different devices.
2. The Flexible Disc Interface drivers which can be redirected to one device, the mass storage device from which the system was booted. (To redirect the Flexible Disc Interface drivers; set **DRV_KEY**, a byte at hexadecimal address **\$FFFFFFEDB**, to 0.)

Only one of the two methods is ever available on any given **BOOT ROM** revision. To determine which is available look at bit 8 of the **BOOT ROM** configuration word (see **BOOT ROM Configuration and Revision** section). If the Read Interface is present, that is the only method available. If the Read Interface is not present, the second method (redirecting the Flexible Disc Interface) is the only method available. (Currently the 3.0 **BOOT ROM** has the Read Interface. The 3.0L **BOOT ROM**, the 1.0 **BOOT ROM**, and the 2.0 **BOOT ROM** do not have the Read Interface.)

This section presents the Read Interface. The section that immediately follows this one, documents the Flexible Disc drivers.

No general purpose disc write drivers are present in the **BOOT ROM** (except for the internal flexible disc). The **BOOT ROM**'s read drivers can be used by any system under the following constraints:

1. The read interface will point to some **msus** (usually the default **msus**, **DEFAULT_MSUS** stored at hexadecimal address **\$FFFFFFEDC**).
2. The read drivers will not concern themselves with interactions with any other I/O drivers. They will not know about multi-tasking, virtual memory, interrupts, running in user mode, or anything over what they do during booting.
3. The read drivers should only be used to do extended booting of additional code.

All routines in the **BOOT ROM** are callable from assembly language directly. Pascal 2.0 in many cases has entry points that handle environment changes and eventually call the routines. See "Using **BOOT ROM** Routines from Pascal". Assembly language calls require that the machine state be in supervisor mode. See trap instructions of **MC68000 User's Manual**, 09826-90073, for putting machine in supervisor mode.

Only one **MSUS** and one file can be open at a time. The routines use the **Error Recovery Block** mechanism for hardware failures or unexpected errors.

All calls to these routines must allocate a memory space for the drivers to utilize. This must be done once only, before making a series of calls to these routines. Below is some assembly language source that shows how to allocate the right amount of memory. A pointer to the allocated block of memory is stored in the bottom of physical memory. The required amount of memory is also specified there.

```

Allocating space:
    mb_size    equ $10           Offset to required size
    mb_ptr     equ $14           Offset to memory pointer
    f_area     equ $FFFD4       Pointer to variables
    movea.l    f_area,a0        Pnt to low RAM variables
    move.l     mb_size(a0),d0    Get required memory
    ..
    <get pointer to d0 bytes of memory in a1>
    ..
    move.l     a1,mb_ptr(a0)     Save pnt to allocated
                                memory

```

The routines can destroy d0-d7 and a0-a4.

The caller must handle any bus error exceptions. Only bus errors caused by a DMA card will be trapped as an error.

The a7 register (Stack pointer) should point to a memory space of at least 1 Kbytes.

If the routines are called from assembly language, a recover block must be set up for handling errors. The following is an example of setting up a recover block:

```

Make a5 point to a recover block.
    link      a5,#-10           allocate block
    pea      recover           set name of error
                                handling routine
    move.l    sp,-10(a5)       save pointer to
                                address of error
                                handling routine
    where recover is the address of a routine that
    extracts the error number from -2 offset off
    register a5 and handles it.

```

There are several generic errors (or escape codes) that can be returned from these routines. They include:

Number	Short Name	Description/Example
-----	-----	-----
1	No Device	Device is missing
2	No Medium	Disc is missing or door is open
3	Not Ready	Controller is busy
4	Read Error	Data lost
5	Bad Hardware	Hardware failure
6	Bad Error State	Software's last ditch effort at error
7	Bus Error	Memory missing

M_INIT

This routine sets up the Read Interface for a mass storage device. It initializes internal temporaries. It determines if the device and media are present and returns true if they are present. It verifies that the device in the parameter msus is of the expected mass storage specifier format. An internal pointer used by M_FOPEN (explained below) is reset to the start of the directory. This routine can escape with any of the possible errors (escape codes): 4 thru 7.

This is usually the first routine to be called in a sequence of calls to the Read Interface.

CALLING FROM PASCAL 2.0 (See "Using Boot ROM Routines from Pascal"):

Declarations:

```
type msustype = packed record {See DEFAULT_MSUS}
    mtype: byte; {Mass Storage Type}
    munit: byte; {Unit Number}
    mscode: byte; {Select Code}
    maddr: byte; {Bus Address}
end;
function boot_init(msus:msustype):boolean; external;
```

CALLING FROM ASSEMBLY LANGUAGE:

default_msus equ \$fffdc	Default mass storage
set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
subq.l #2,sp	Reserve space for boolean
move.l default_msus,-(sp)	Pass MSUS parameter
jsr \$4004	Call M_INIT
tst.b (sp)+	Test returned result
clear error recovery	(arguments are removed by the routine)

M_FOPEN

This routine attempts to open the specified file, parameter filename below. If successful it returns the following: return true, the actual file name opened, the actual file type, the size in bytes and start execution address in memory. If successful the routine also sets up the Read Interface driver to read sectors relative to the opened file. If a character of the file name is a null, then a wild card search of anything with the same previous characters is done. If filetype (ftype) is -1, it will allow any file type. Otherwise, the filetype must also match. (Any file type is allowed.) The first found is used. M_FOPEN returns false if the device or media is not present. The routine can escape with any error (or escape code). Searching for a file always begins at the point left off by the last call to M_FOPEN. (SRM is an exception. When a specific file name on an SRM is given with no wild card, it will try to open it directly without affecting the current search state.)

When the mass storage specifier is the Shared Resource Manager the file name is actually a path name into a directory tree of the following format:

/n1<p1>/n2<p2>/n3<p3>/n4<p4>/n5<p5>/n6<p6>

Where:

/ is a separator between path name parts,

n1 thru n6 are names that can be up to 16 characters each in length,

<p1> thru <p6> are optional passwords of up to 16 characters each in length,

/n1<p1>/ can be left off to imply "/SYSTEMS/", and

everything to the right of and including any "/" is optional. This means that files can be found at any of 6 different tree levels.

This routine is typically called after another M_FOPEN call to do directory searching. M_INIT must be called before the first call to M_FOPEN.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
type string255 = string[255]; {Maximum possible string}
     shortint = -32768..32767; {16-bit integer}
function boot_mfopen(var filename:string255;
                    var x_adr,length:integer;
                    var ftype:shortint):boolean; external;
```

Where:

filename is the file name. x_adr is the returned execution address. length is the size in bytes of the file. ftype is the file type of the file opened.

CALLING FROM ASSEMBLY LANGUAGE:

set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
subq.l #2,sp	Reserve space for boolean
pea filename	Pass file name pointer
pea x_adr	Pass execution addr. ptr.
pea length	Pass length pointer
move #-1,ftype	Set file type
pea ftype	Pass file type pointer
jsr \$4008	Call M_FOPEN
tst.b (sp)+	Test returned result
clear error recovery	(arguments are removed by the routine)

M_READ

M_READ reads the specified number of bytes (bytecount) starting at the specified sector (a sector is 256 bytes) to a specified RAM location (ramaddress). Sector number is relative to a file if one is open and if the passed parameter, "media" is set to false, absolute on media otherwise. (Sector zero is the first sector.) Bus error exception is handled by caller.

If file relative reading is being done (media is false), M_FOPEN must have been called sometime previously to open the file. M_INIT must have been previously called before any M_READ calls can be made.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
function boot_mread(sector,bytecount,ramaddress:integer;
                    media:boolean):boolean; external;
```

Where:

sector is the sector number to start reading. bytecount is the number of bytes to read. ramaddress is the location in memory to transfer the data read. media is a boolean that determines if the reading to be done is relative to the media.

CALLING FROM ASSEMBLY LANGUAGE:

set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
subq.l #2,sp	Reserve space for boolean
move.l sector,-(sp)	Pass the sector number
move.l bytecount,-(sp)	Pass the no. of bytes
move.l ramaddress,-(sp)	Pass destination address
move.w media,-(sp)	Pass media relative bool.
jsr \$400C	Call M_READ
tst.b (sp)+	Test returned result
clear error recovery	(arguments are removed by the routine)

M_FCLOSE

This routine closes any file previously opened by M_FOPEN. This is typically the last routine called in a sequence of calls to the Read Interface. M_FOPEN must have been called previously.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
function boot_mfclose; external;
```

CALLING FROM ASSEMBLY LANGUAGE:

```
set error recovery          (see F_PWR_ON in Flexible
                             Disc Driver section)
jsr      $4010              Call M_FCLOSE
clear error recovery        (arguments are removed by
                             the routine)
```

This completes the presentation of the Read Interface. The Read Interface is only intended for use as a part of an extended load operation. The next section presents the Flexible Disc Drivers. They must be used in the absence of the Read Interface for extended loads. The Flexible Disc Drivers are also intended to be used by anyone who wants to talk to the built-in flexible disc drive(s).

A typical sequence of calls to the read interface would be:

```
M_INIT   to point the read interface at a disc
M_FOPEN  to open a file for reading
M_READ   multiple calls, to read data into memory
M_FCLOSE to clean-up and close files
```

Flexible Disc Drivers

The BOOT ROM contains a set of device drivers for the internal 5.25 inch flexible disc drive(s). This code is actually an extension of the hardware design and provides for hardware timing support. Thus, it is recommended that these drivers be used before any consideration is made to talk to the flexible disc hardware directly.

When a BOOT ROM revision has no Read Interface, it is necessary to utilize the read drivers of this interface to do any loading operation. This is done by setting the byte DRV_KEY at hexadecimal address \$FFFEDB, to zero. (In normal operation, DRV_KEY should be set to true or non-zero.)

If there is no flexible disc present, these routines will escape with the error value (escape code) of 2080. (DRIVE, a byte at hexadecimal address \$FFFFFFED3, should be set to 0 or 1 to select drives 0 and 1, respectively.) (See High RAM Map for other variables used.)

Below is an enumeration of all the possible errors (escape codes) that the flexible disc drivers can return: (The first seven can only occur when the flexible disc driver interface is redirected with DRV_KEY set to 0.)

```

1   No Device
2   No Medium
3   Not Ready
4   Read Error
5   Bad Hardware
6   Bad Error State
7   Bus Error
1066 bad track 0, side 0
2066 more than 4 spares
3066 write fault or lost data
4066 timeout during initialize
1080 no media or door open
2080 no media or door open
3080 no media or door open
8080 media changed
9080 media changed during operation
1081 track not found
2081 restore error
3081 track 0 not found after reset
4081 read lost data error
5081 write lost data error or fault
6081 address lost data error
7081 address CRC error during write
1083 write protect error
2083 write protect error
1084 read record not found/d bit set
2084 write record not found
3084 address (track) not found
1087 address CRC error
1088 read CRC error
6090 unexpected interrupt
7090 interrupt during write track handshaking
8090 timeout waiting for interrupt
9090 interrupt mask > 2 (drive locked out)
11090 timeout; drive not responding
1082 2nd drive not present

```

The error reporting mechanism for all of the flexible disc drivers is as follows:

```

link      a5,#-10           Make error recover block
move.w    #error no.,-2(a5) set error number
move.l    -10(a5),sp       set the stack pointer
rts                               'return' to the
                                   recovery routine

```

This mechanism is compatible with Pascal 1.0 directly. Pascal 2.0 has a slightly different mechanism. Pascal 2.0 has separate entries to all routines to handle differences (see "Using Boot ROM Routines from Pascal").

All routines in the BOOT ROM are callable from assembly language directly. (Pascal 2.0 in many cases has entry points that handle environment changes and eventually call the routines. Assembly language calls require that the machine state be in supervisor mode. See trap instructions of MC68000 User's Manual, 09826-90073, for putting machine in supervisor mode.)

F_PWR_ON / RESET

This routine sets the level 2 interrupt vector, initializes the High RAM variables used by the drivers and resets the drive. This routine must be called at least once before any calls to other flexible disc drivers or file utilities are made.

F_PWR_ON (Flexible Disc Power On) is called by the BOOT ROM; before any system is started; so that it is only required to be called by a system after a RESET instruction has been executed.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
procedure asm_f_pwr_on; external;
```

CALLING FROM ASSEMBLY LANGUAGE:

```
*
* Set-up an Error Recover Block
*
    link    a5,#-10           Make error recover block
    pea    recover           Push error handler addr.
    move.l  sp,-10(a5)       Push stack pointer
*
* Call the Routine
*
    jsr    $144              Call F_PWR_ON
*
* Clear Error Recover Routine
*
    unlk   a5                Remove recover block from
                           the stack (any arguments
                           are removed by routine)
```

ASSEMBLY LANGUAGE ERROR RECOVERY ROUTINE:

```
*
* This is the recover routine that goes with the above
* assembly code. This routine is the same for all flexible
* disc routines. At the Pascal language level this is taken
* care of automatically via try-recover. (Try-recover is the
* mechanism for trapping errors.)
recover equ    *
    move.w    -2(a5),d0      Get the error number
    unlk     a5              Clean off the stack
    process  escape code
```

FLPYREAD

This routine reads a specified 256 byte sector from the drive into a given RAM location. (FUBUFFER at hexadecimal address \$FFFFD2 is a 256 byte buffer that can be used as a destination for a sector of data.)

Arguments are not range checked. A bus error could occur because of an invalid buffer address. F_PWR_ON must have been called some time previously.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
procedure asm_flpyread(sector:integer;
                       var buffer:integer); external;
```

Where:

sector is the sector number to read into memory at address pointed to by buffer. A sector is 256 bytes. The legal range for sector number is 0 to 1055 for the internal flexible disc.

CALLING FROM ASSEMBLY LANGUAGE:

```
set error recovery          (see F_PWR_ON in Flexible
                             Disc Driver section)
move.l    sector, -(sp)    Set sector number
pea      BUFFER           Set buffer address
jsr      $120             Call FLPYREAD
clear error recovery       (arguments are removed by
                             the routine)
```

FLPY_WRT

This routine writes a specified 256 byte sector from a given RAM location on the disc.

Arguments are not range checked. An invalid buffer address will cause a bus error. F_PWR_ON must have been previously.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
procedure asm_flpy_wrt(sector:integer;
                       var buffer:integer); external;
```

Where:

sector is the sector number to write using the data in memory at address pointed to by buffer. A sector is 256

bytes. The legal range for sector number is 0 to 1055 for the internal flexible disc.

CALLING FROM ASSEMBLY LANGUAGE:

set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
move.l sector, -(sp)	Set sector number
pea BUFFER	Set buffer address
jsr \$124	Call FLPY_WRT
clear error recovery	(arguments are removed by the routine)

FINTRUPT

This routine is the ISR (Interrupt Service Routine) for the flexible disc drivers. It resets the flexible disc interrupt bit. It tests and clears bit 1 of FFLAGS. (FFLAGS is an internal temporary variable to the Flexible Disc Drivers. It does not need to be accessed directly. The other Flexible Disc drivers automatically manipulate FFLAGS correctly.) Bit 1 of FFLAGS must be set when the interrupt occurs. (If it was not set, then an error results.) Bit 2 of FFLAGS is set by FINTRUPT. F_PWR_ON must have been called previously.

USING ASSEMBLY LANGUAGE TO SET LEVEL 2 VECTOR:

set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
move.w \$4ef9, \$ffffffb8	Move JMP to vector
move.l #\$128, \$ffffffba	Move addr of FINTRUPT

FLPYINIT

This routine initializes the internal flexible disc. The interleave factor is one word (16 bits). The address of the CRT message area is assumed to be even and is intended to be in the CRT alpha area. The message is written to the odd bytes. The message is:

INITIALIZE: TRACK tt, SIDE s, SPARED n

Arguments are not range checked. A bus error could result from a bad message address. Interleave is taken modulo 16. F_PWR_ON must have been called previously.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
type shortint = -32768..32767;
procedure asm_flpyinit(crtptr: anyptr;
                      interleave: shortint); external;
```


Where:

crtptr is a valid pointer into CRT memory. (See other sections of manual for range of CRT memory.)
interleave is a value in the range of 1 to 15.

CALLING FROM ASSEMBLY LANGUAGE:

set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
move.l crtptr,-(sp)	Set CRT message line ptr
move.w interleave,-(sp)	Set interleave factor
jsr \$12C	Call FLPYINIT
clear error recovery	(arguments are removed by the routine)

FLPYMREAD

This routine reads a given number of sectors into a given RAM location beginning at a specified sector. Arguments are not range checked. A bus error could occur because of an invalid buffer address. F_PWR_ON must have been called previously.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
procedure asm_flpymread(sector_count,sector:integer;  
                        var buffer:integer); external;
```

Where:

sector is the first sector number to start reading into memory at address pointed to by buffer. A sector is 256 bytes. The legal range for sector number is 0 to 1055 for the internal flexible disc. sector_count is the number of contiguous sectors to read.

CALLING FROM ASSEMBLY LANGUAGE:

set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
move.l sec_cnt,-(sp)	Set number of sectors
move.l sector,-(sp)	Set sector number
pea buffer	Set buffer address
jsr \$130	Call FLPYMREAD
clear error recovery	(arguments are removed by the routine)

FLPYMWRITE

This routine writes a given number of sectors from a given RAM location beginning at a specified sector. Arguments are not range checked. An invalid buffer address will cause a bus error. F_PWR_ON must have been called previously.

CALLING FROM PASCAL 2.0 (see "Using Boot ROM Routines from Pascal"):

Declarations:

```
procedure asm_flpymwrite(sector_count,sector:integer;
                        var buffer:integer); external;
```

Where:

sector is the first sector number to start writing using data in memory at address pointed to by buffer. A sector is 256 bytes. The legal range for sector number is 0 to 1055 for the internal flexible disc. sector_count is the number of contiguous sectors to write.

CALLING FROM ASSEMBLY LANGUAGE:

```
set error recovery          (see F_PWR_ON in Flexible
                             Disc Driver section)
move.l    sec_cnt,-(sp)     Set number of sectors
move.l    sector,-(sp)     Set sector number
pea      buffer            Set buffer address
jsr      $134              Call FLPYMWRITE
clear error recovery       (arguments are removed by
                             the routine)
```

FMSGs

This routine formats the text for an error message. It is specifically designed for flexible disc error codes. (Read Interface errors, 1 thru 7, will not be displayed mnemonically.)

The format is:

```
{number MOD 1000},{number DIV 1000} text
```

e.g. an 8080 value would result in:

```
80,8 MEDIA CHANGED
```

If the error number does not match an entry in the text table, only the number is printed.

On entry to the routine, register D0 (least significant 16 bits) contains the error code. Register A1 contains the address to which the message is to be written (RAM not CRT).

CALLING FROM ASSEMBLY LANGUAGE:

```

        lea     buffer,a1           Set the buffer pointer
        move.w  escapecode,d0      Set the error number
        jsr     $1BC               Call the formatter
*
* Then to put on CRT (Example)
*
        lea     buffer,a0          Point to message
        moveq   #1,d0              Put on line 1 of CRT
        jsr     crtmsg             Call routine documented
                                   in later section

```

This concludes the presentation of the Flexible Disc Drivers. The next section presents routines that can be used to cause the BOOT ROM to load and start another system.

System Switching

In some applications, the strengths of multiple operating systems may be important. The BOOT ROM provides a very low level method for automatically switching between operating systems. The BOOT ROM allows one operating system to replace itself with another operating system. This is called system switching.

The routines in this section present three ways to start another system. The first two have minimal environment requirements. The last of the three requires the same environment as the Read Interface presented in an earlier section. (All three routines leave the machine in state specified by section "CPU State at Load".)

REQ_BOOT

This routine, REQ_BOOT (request boot), will load and start a system. SYSNAME (at hexadecimal address \$FFFFDC2) is a 10 byte location that contains the file name of the system to start. If the first character is null (binary zero) then the powerup search sequence will be made for a system. If the first character is not null and the second one is, then a search will be made for soft system "SYSTEM_Cxx" where C is the given character and xx are any characters. If the search fails then ROM system C will be searched for. (On 3.0 and later BOOT ROMs, the DEFAULT_MSUS chooses mass storage device or ROM. Also, a character followed by a null will cause a wild card search for the first system that starts with the character.) If neither the first or second character is null then only soft systems will be searched for.

BOOTTYPE (a 16 bit word at hexadecimal address \$FFFFDC0) will be set to 18 (BOOTTYPE=18 indicates that a boot or system switch was done) when the system starts and the machine will be in power-up state.

CALLING FROM ASSEMBLY LANGUAGE:

```

sysname    equ $ffffdc2           10 Character System Name
move.l     'SYSQ',sysname         Set the name
clr.b      sysname+4              Set the terminator
jmp        $1C0                   Do it

```

REQ_REBOOT

This routine, REQ_REBOOT (request reboot), is exactly the same as REQ_BOOT except that BOOTTYPE will be set to 12 (BOOTTYPE=12 indicates that a re-boot was done) and the routine entry point is at hexadecimal address \$1A4.

BOOT

The 3.0 BOOT ROM and later BOOT ROMs present another interface (when the Read Interface is present). This interface allows the specification of up to a 255 character file name. The file name has the same format as M_FOPEN presented with the Read Interface. The new interface also will return if the device is inaccessible or the file cannot be opened.

If the system is not found, it will return to the caller. If it is successful, it will never return to the caller. There are several cases where returning is impossible because the caller or some of the caller's data structures may have been destroyed by the booting process. In these cases, it will never return to the caller.

The booter will return to the caller if the mass storage device is not present, the system is not found, or requested boot is not a valid system. If the boot error is unrecoverable the booter will display an error message on the line next to the bottom of the screen and hang.

The caller has the option of setting up the level 7 interrupt vector before calling the routine.

Before calling the routine, the environment specified for the Read Interface needs to be set up. This includes allocating space for the mass storage drivers (as shown below), and changing the machine state to supervisor mode. Currently Pascal 2.0 is the only language which is not already in supervisor mode. (See trap instructions of MC68000 User's Manual, 09826-90073, for switching to supervisor mode.)

ALLOCATED SPACE FROM ASSEMBLY LANGUAGE:

```
mb_size    equ $10           Offset to required size
mb_ptr     equ $14           Offset to memory pointer
f_area     equ $FFFED4       Pointer to variables
movea.l    f_area,a0        Pointer to low RAM
*
move.l     mb_size(a0),d0    Get required memory
..
<get pointer to d0 bytes of memory in a1>
..
move.l     a1,mb_ptr(a0)    Save pointer to allocated
*                               memory
```

CALLING FROM PASCAL 2.0:

(see "Using Boot ROM Routines from Pascal"): PASCAL Declarations:

```
type msustype = packed record {See DEFAULT_MSUS.}
    mtype: byte; {Mass Storage Type}
    munit: byte; {Unit Number}
    mscode: byte; {Select Code}
    maddr: byte; {Bus Address}
end;
```

```
    string255 = string[255];  
function boot(msus:msustype;  
             var filename:string255); external;
```

Assembly Language Declarations:

```
def      boot  
boot    equ $4000
```

CALLING FROM ASSEMBLY LANGUAGE:

default_msus equ \$ffedc	Default mass storage
set error recovery	(see F_PWR_ON in Flexible Disc Driver section)
subq.l #2,sp	Reserve space for boolean
move.l default_msus,-(sp)	Pass MSUS parameter
pea filename	Pass file name
jsr \$4000	Call BOOT
jmp *	Boot failed if it returns
clear error recovery	(arguments are removed by the routine)

This concludes the presentation of the routines used for switching to another system from inside an operating system. The next sections present miscellaneous BOOT ROM routines that are available for use.

CRTINIT

This routine sets the CRT controller registers 0 thru 11. These are the registers that control the actual operation of the horizontal and vertical sweep circuitry. This routine is automatically called by the BOOT ROM at power-up and should not be necessary at a later time. But, some operating systems inadvertently modify these registers while they are attempting to move the CRT cursor. This routine can be used to put the CRT controller back to its power-on settings.

Calling CRTINIT from Pascal 2.0 requires that the machine be put in supervisor mode first. (See MC68000 User's Manual, 09826-90073, regarding trap instructions that do this.)

CALLING FROM PASCAL 2.0:

(see "Using Boot ROM Routines from Pascal"):

PASCAL Declarations:

```
procedure crtinit; external;
```

Assembly Language Declarations:

```
def      crtinit
crtinit  equ $13C
```

CALLING FROM ASSEMBLY LANGUAGE:

```
jsr      $13C          Call CRTINIT
```

If initializing the CRT controller is desired, it is recommended that CRTINIT be used rather than attempting the operation from an operating system application. The next section presents some CRT operations that are recommended for an operating system application.

CRTCLEAR/CRTMSG

Two routines that commonly appear in other Internals Manual examples are CRTCLEAR and CRTMSG. Unfortunately, these two routines do not exist in the same form in all of the revisions of the BOOT ROM. So it is advised for a user to code these routines himself. Fortunately they are so simple that the source code is included below. CRTCLEAR clears the CRT. CRTMSG displays one line of text on a specified line of the CRT.

```
*****
* CRTCLEAR -- Initializes the CRT, then blanks it.          *
* CRTBLANK -- Only blanks the CRT.                          *
*   CRTCLEAR requires supervisor mode.                      *
*   CRTBLANK can be called directly from assembly language *
*       in any environment.                                  *
*****
*
* Declarations
*
crtmb     equ     $5121a0      Beginning of alpha
crtendb   equ     $513140      End of alpha
crtinit   equ     $13C        Address of CRTINIT
*
* CRTCLEAR routine
*
crtclear  jsr     crtinit      Initialize the CRT
*
* CRTBLANK routine
*
crtblank  move.l  #$00200020,d0  Blank CRT memory
          lea     crtmb,a0       Start at beginning
clrloop   move.l  d0,(a0)+      Clear two at a time
          cmpa.l  #crtendb,a0    Stop at end of CRT mem.
          blt     clrloop
          rts
```

```

*****
* CRTMSG -- Write message to given line of CRT. *
* Can be called directly from assembly language in any *
* environment. *
* *
* On entry: a0 - contains address of null terminated *
* message to display on CRT. *
* d0 - contains CRT line number to place *
* message where line 0 is the top *
* line of the CRT. *
*****
*
* Declarations
*
crtma equ $512704 Beginning of 9826 alpha
crtmb equ $5121a0 Beginning of 9836 alpha
crtllena equ 100 2 X 9826 CRT line length
crtllenb equ 160 2 X 9836 CRT line length
*
* CRTMSG routine
*
crtmsg btst #0,sysflag 80 or 50 wide alpha?
      beq.s crtmsgb 80 wide
      mulu #crtllena,d0 50 wide so *100
      add.l #crtma,d0 Use 9826 alpha offset
      bra.s crtmsg0 Go to copy string
crtmsgb mulu #crtllenb,d0 80 wide so *160
      add.l #crtmb,d0 Use 9836 alpha offset
crtmsg0 movea.l d0,a1 Use a1 to point to alpha
      moveq #0,d0 Clear highlight byte
crtmsg1 move.b (a0)+,d0 Get a character of string
      beq.s crtreturn If null, then done
      move.w d0,(a1)+ Put character on screen
      bra.s crtmsg1 Get next character
crtreturn rts Done

```


NMI__DECODE

NMI__DECODE is the address of the NMI Interrupt Service Routine (ISR) used by the BOOT ROM. (An NMI is a level 7 non-maskable interrupt.) It determines which device caused the NMI then jumps to one of four (4) pseudo vectors:

FFFFFF34	PSEUDO VECTOR 1	RESET from keyboard
FFFFFF2E	PSEUDO VECTOR 2	Keyboard timer timeout (fast handshake)
FFFFFF28	PSEUDO VECTOR 3	Battery backup interrupt
FFFFFF22	PSEUDO VECTOR 4	NMI from the backplane

The address at location \$1B0 must be moved to the level 7 vector. The variable, BATTERY, must be set correctly. This is done by boot code.

After initializing the level 7 vector to use this routine the above four vectors can be initialized to jump to operating system dependent routines that are terminated by the RTE instruction. (See *MC68000 User's Manual*, 09826-90073, for more detail.)

EXAMPLE FROM ASSEMBLY LANGUAGE:

```
*
* Setting Level 7 Vector
*
    move.w    #$4ef9,$ffffff9a    Set JMP opcode
    move.l    $1b0,$ffffff9c     Set the ISR address
```

CRASH

This routine is intended to recover the system after an interrupt or pseudo vector not setup by the operating system occurs.

Once called, CRASH displays the message (before hanging):

```
UNEXPECTED USE OF aaaaaaaa
```

Where aaaaaaaa is the address in hexadecimal of the JSR. This address is computed as return address minus 6. The boot code loads all vector locations with JSR CRASH prior to the system being given control.

EXAMPLE FROM ASSEMBLY LANGUAGE:

```
jsr      $1b8          Go crash
```

This concludes the presentation of all routines that the BOOT ROM provides.

Character Table

The BOOT ROM also makes available a readable copy of the CRT character ROM for the 9826.

The raster character patterns for the 9826A CRT are stored in addresses \$2000 to \$2FFF. These patterns are stored 16 bytes per character, with each byte representing 8 horizontal pixels. The first byte is the upper 8 pixels of the character, and bit 7 is the leftmost pixel of the byte. The characters are in order by character code with 0 first and 255 last.

To save on cost, the 3.0L BOOT ROM does not have valid data at these addresses. See section on BOOT ROM Configuration Identification to determine if the BOOT ROM present has a valid character table.)

To round out the presentation of the BOOT ROM, the next two sections present memory maps of both the low ROM and high RAM areas.

High RAM Map

The high RAM map specifies soft vectors that are at the disposal of operating systems to process exceptions. The map specifies and summarizes variables used or set-up by the BOOT ROM. The map also shows where memory can be safely accessed by a system.

The layout of high memory usage by the BOOT ROM follows. It starts at top of memory and works downward.

The following vectors are accessed via the 68000 exception vectors in low ROM. 68000 exceptions transfer control to these RAM vectors which normally have been initialized by the language system.

```
*
*   Soft Interrupt Vectors
*
FFFFFFFA   Bus Error
FFFFFFF4   Address Error
FFFFFFEE   Illegal Instruction
FFFFFFE8   Divide By Zero Trap
FFFFFFE2   CHK Instruction Trap
FFFFFFDC   TRAPV Instruction Trap
FFFFFFD6   Privilege Instruction Violation
FFFFFFD0   Trace Trap
FFFFFFCA   1010 Opcode Line Emulator
FFFFFFC4   1111 Opcode Line Emulator

FFFFFFBE   Interrupt Level 1
FFFFFFB8           2
FFFFFFB2           3
FFFFFFAC           4
FFFFFFA6           5
FFFFFFA0           6
FFFFFF9A           7

FFFFFF94   TRAP Instruction 0
FFFFFF8E           1
FFFFFF88           2
FFFFFF82           3
FFFFFF7C           4
FFFFFF76           5
FFFFFF70           6
FFFFFF6A           7
FFFFFF64           8
FFFFFF5E           9
FFFFFF58           A
FFFFFF52           B
FFFFFF4C           C
FFFFFF46           D
FFFFFF40           E
FFFFFF3A           F
```

The ROM has a facility for decoding NMI; if the language system routes NMIs thru NMI_DECODE, it will jump to one of the following four vectors depending on what caused the NMI:

FFFFFF34	Pseudo Vector 1	RESET from keyboard
FFFFFF2E	Pseudo Vector 2	Keyboard timer timeout (fast handshake)
FFFFFF28	Pseudo Vector 3	Battery backup interrupt
FFFFFF22	Pseudo Vector 4	NMI from the backplane
FFFFFF1C	Spurious Interrupt	
FFFFFF16	Vectored	" 0
FFFFFF10	"	" 1
FFFFFF0A	"	" 2
FFFFFF04	"	" 3
FFFFFFFE	"	" 4
FFFFFFF8	"	" 5
FFFFFFF2	"	" 6
FFFFFFEC	"	" 7
FFFFFFE6	Pseudo Vector	(unassigned) These have no stated
FFFFFFE0	"	" purpose
FFFFFFDC	DEFAULT_MSUS	Default Mass Storage Specifier

VARIABLES:

FFFFFFDB	DRVR_KEY	Disable Reads to Default MSUS Thru Flexible Disc (Ignored by 3.0 BOOT ROM - Use READ Interface)
FFFFFFDA	SYSFLAG2	This is a system configuration byte.
FFFFFFD9	RETRY	Used by Flexible Disc Drivers
FFFFFFD8	NDRIVES	This byte is the number of internal drives minus one (255 means no drives) (Not Valid on 1.0 BOOT ROMs)
FFFFFFD4	F_AREA	This long-word points to an area in low RAM that is used for mass storage temporaries. The address of the first word above this area is contained in LOWRAM. (F_AREA is not valid on 1.0 BOOT ROMs)
FFFFFFD3	DRIVE	Language systems set this byte to indicate the drive (0 or 1) to be accessed with the next flexible disc operation. (Not Valid on 1.0 BOOT ROMs)

```

FFFFFED2  SYSFLAG  This is a system configuration byte.
FFFFFDD2  FUBUFFER  This is a 256 byte buffer for use by mass
              storage drivers
FFFFFDCE  LOWRAM   When control passes from the booter to the
              operating system, this location will contain
              the address of the lowest byte of usable RAM
              (four bytes).
FFFFFDCD  BATTERY  This byte will contain a 1 if battery backup
              is installed; 0 otherwise.
FFFFFDC2  SYSNAME  This 10 byte area will contain the ID byte for
              the ROM being started or the name of the file
              the system was loaded from.  If the file name
              is less than 10 bytes long, the last byte +1
              MUST BE binary 0.  (For example is the file
              name is 5 characters, then character number 6
              should be a zero.)  This convention applies
              when boot code is invoked to switch systems.
FFFFFDC0  BOOTTYPE This word will contain a code to indicate what
              condition caused the system to start.
              0 = power on
              12 = re-boot requested
              18 = boot or system switch
FFFFFDBC  STARTADDRESS  For soft systems, this byte contains the
              start address of the system.  For ROM systems,
              this byte is meaningless.
FFFFFDBC  SYSTEMS SHOULD BE CAREFUL OF USING MEMORY AT THIS
              ADDRESS OR HIGHER FOR TEMPORARIES.

-----
other booter temporary variables
-----
FFFFFDAC  Top of Booter stack
-----
FFFFFAC0  SYSTEMS SHOULD NEVER LOAD OVER THIS LOCATION OR HIGHER.
              The 3.0 BOOT ROM will check this, previous ones do not.
-----
FFFFF9A4  SYSTEMS SHOULD NOT LOAD OVER THIS LOCATION OR HIGHER,
              If they want SYSNAME to be preserved throughout the
              load.

```

Low ROM Map – Exception Vectors

The Low ROM MAP (hard coded in the BOOT ROM) specifies where to jump for the various exceptions and routines.

Addr.	Contents

0000	Power up RESET stack address
0004	Power up RESET start address (This goes to the BOOT ROM.)
0008	FFFFFFFA Bus Error
000C	FFFFFFF4 Address Error
0010	FFFFFFFE Illegal Instruction
0014	FFFFFFE8 Divide By Zero Trap
0018	FFFFFFE2 CHK Instruction Trap
001C	FFFFFFDC TRAPV Instruction Trap
0020	FFFFFFD6 Privilege Instruction Violation
0024	FFFFFFD0 Trace Trap
0028	FFFFFFCA 1010 Opcode Line Emulator
002C	FFFFFFC4 1111 Opcode Line Emulator
0030 - 005F	Unassigned Vectors (Motorola has reserved these)
	00000000
0060	FFFFFF1C Spurious Interrupt
0064	FFFFFFBE Interrupt Level 1
0068	FFFFFFB8 " " 2
006C	FFFFFFB2 " " 3
0070	FFFFFFAC " " 4
0074	FFFFFFA6 " " 5
0078	FFFFFFA0 " " 6
007C	FFFFFF9A " " 7
0080	FFFFFF94 TRAP Instruction 0
0084	FFFFFF8E 1
0088	FFFFFF88 2
008C	FFFFFF82 3
0090	FFFFFF7C 4
0094	FFFFFF76 5
0098	FFFFFF70 6
009C	FFFFFF6A 7
00A0	FFFFFF64 8
00A4	FFFFFF5E 9
00A8	FFFFFF58 10
00AC	FFFFFF52 11
00B0	FFFFFF4C 12
00B4	FFFFFF46 13
00B8	FFFFFF40 14
00BC	FFFFFF3A 15

00C0 -- 00FF	
	00000000

```

0100 FFFFFFF16      Vectored Interrupt 0 (User Interrupt Vector)
0104 FFFFFFF10      1
0108 FFFFFFF0A      2
010C FFFFFFF04      3
0110 FFFFFFFFE      4
0114 FFFFFFFE8      5
0118 FFFFFFFE2      6
011C FFFFFFFE0      7

```

```

0120 JMP Flexible Disc Sector Read & move data to user buffer
0124 JMP Flexible Disc Sector Write from user buffer
0128 JMP Flexible Disc ISR
012C JMP Flexible Disc Initialize LIF disc
0130 JMP Flexible Disc Multi-sector Read to user buffer
0134 JMP Flexible Disc Multi-sector Write from user buffer
0138 Internal routine
013C JMP CRTINIT Initializes CRT registers R0 - R11
0140 Internal routine
0144 JMP Flexible Disc Power On

```

```

-----
0148 -- 01A3      Internal routines
01A4 JMP REQ_REBOOT Re-load and start current system
01A8 -- 01AF      Internal routines
01B0 address NMI_DECODE Address of NMI decoder ISR
01B4 Internal routine
01B8 JMP CRASH      Crash recovery routine
01BC JMP FMSGs      Flexible Disc error messages routine
01C0 JMP REQ_BOOT   Boot SYSNAME system
01C4 -- 01F3      Internal routines and Vectors

```

=====

User Interrupt Vectors (125-255) (New as of 3.0 BOOT ROM):
(No hardware currently supports user interrupt vectors.)

```

01F4 FFFFF400      User Interrupt Vector 125
01F8 FFFFF406      User Interrupt Vector 126
01FC FFFFF40C      User Interrupt Vector 127
0200 FFFFF412      User Interrupt Vector 128
0204 FFFFF418      User Interrupt Vector 129
0208 FFFFF41E      User Interrupt Vector 130
. . . . .
. . . . .
03F8 FFFFF706      User Interrupt Vector 254
03FC FFFFF70C      User Interrupt Vector 255

```

Using Boot ROM Routines from Pascal

Most of the routines in the Boot ROM are not really designed to be used after the booting process is complete. In particular, these routines don't address variables or report errors in a way which is fully compatible with Pascal 2.0; and Boot ROM routines expect to run in Supervisor mode, while user programs in Pascal 2.0 run in User mode.

The Pascal OS itself uses an assembly language module called ROMCALL to provide the necessary interfacing. This module provides Pascal-callable entry points for some commonly used routines, and may be used as an example of how it's done. The purpose of this section is to give some guidelines for you to follow if you want to call other routines yourself.

1. Routines in the Boot ROM will operate properly only in supervisor mode. In the PASCAL 2.0 system, entry into supervisor mode can be forced with a TRAP #11 instruction. The exception handler for TRAP #11 leaves the old status register (SR) contents on the stack, so that the previous mode can be recovered easily.

For example,

```
TRAP #11      enter supervisor mode
...          make whatever calls must be done in supervisor
MOVE (SP)+,SR restore status register, hence initial mode
```

2. Since switching from user mode to supervisor mode implies changing which stack A7 points to, any parameters which are passed on the stack must be transferred to the other stack whenever a switch is made. One way to do this is move the parameters into registers while the switch is done, then back to the (other) stack after the switch. The TRAP #11 exception routine uses no registers. Remember that the return address is on the stack in front of any parameters if your routine provides an interface to the Boot ROM.

```
MOVEM.L (SP)+,D0-D4   move 4 integer parameters off the stack
MOVE.L D0,-(SP)      replace return address onto stack
TRAP #11             switch stacks, SR saved on stack
MOVEM.L D1-D4,-(SP)  put parameters on other stack
...                 make necessary call
MOVE (SP)+,SR        restore stack, mode
RTS                  return to caller
```


3. If the Boot ROM routine accesses the internal mini floppy disk drive, you must change the level 2 interrupt vector before calling the routine. (The following routine assumes that it is not necessary to save and restore the old contents of the level 2 interrupt vector. Usually the save/restore IS necessary, and it's up to you to figure out where you want to save it.)

```

INTVEC2 EQU -72          absolute address of level 2 interrupt
                          vector
FINTRUPT EQU $128       Boot ROM interrupt service routine
                          address

LEA    INTVEC2,A3       point to level 2 interrupt vector
MOVE.W #20217,(A3)+     move a JMP instruction to vector
MOVE.L #FINTRUPT,(A3)   move address of Boot ROM ISR to vector
...                               make necessary calls to Boot ROM

```

4. If the Boot ROM routine being called has any error conditions, it will escape using a mechanism similar to the TRY - RECOVER construct used by PASCAL 2.0. The approximate sequence that the Boot ROM will use is:

```

MOVE.W #ERRORCODE,-2(A5)   save "escapecode"
MOVEA.L -10(A5),SP         pop stack to "recover block"
RTS                        execute error recovery code

```

This sequence is not compatible with the PASCAL 2.0 workstation recover method, although it was compatible with the PASCAL 1.0 workstation. One possible sequence to set up this error recovery environment follows:

```

LINK    A5,#-2           save old A5 on stack, save space for
                          error code
PEA     RECOVER           -6(A5) address code to be executed on
                          error
PEA     (SP)              -10(A5) address "recover block" on
                          stack
...                               call Boot ROM
UNLK    A5                normal return, no error, pop
                          "recover block"
...

* errors return here:
RECOVER MOVE.W (SP)+,errorcode
                          (equivalent to MOVE.W -2(A5),error code)
MOVEA.L (SP)+,A5
                          (equivalent to UNLK A5 )
...                               process the error

```

The example given above will trap any errors generated by the Boot ROM, but will not trap the stop or clear I/O keys on the PASCAL 2.0 workstation.

Creating a Bootable System

You can use the Assembler, Pascal Compiler and Librarian to create a bootable disc with a standalone application or operating system on it. This is may be very important to a few customers, but is to be avoided by most people because it's probably a lot of work in the end.

Before giving some guidelines and an example, it seems appropriate to remind you of some of the reasons to avoid writing your own bootable system:

1. You will have to write physical drivers for IO devices such as discs. This is not easy to do. You could ruin an expensive drive by seeking the head into the spindle, which makes a horrible noise. Also, you'd need documentation on the disc protocols.
2. You'll probably need a file system. If you invent a new, incompatible file system you'll end up with applications that can't communicate with anything else -- your own narrow little world.
3. You can take advantage of the drivers and other software in the Pascal system by using our kernel and replacing the STARTUP program (the Command Interpreter) with your application. This will have most of the benefits of writing your own operating system, without most of the pain.

Verbum Sapienti.

Guidelines for System Creation

When creating your own system, you are entirely responsible for all aspects of the environment in which you run. There is no operating system unless you implement it. You must be thoroughly familiar with the M68000 processor and the other resources provided by your machine. If you are using the PASCAL compiler to generate code, then you must be familiar with the code produced by the compiler and the internal conventions it follows.

The things you must do include but are not limited to the following:

1. Decide where your code is to be loaded into memory (see BOOT command).
2. Set your stack pointer to an appropriate place.
3. Set up all interrupt vectors which your system will use.
4. Set up all vectors for exceptions which might occur in your system.
5. Set up handlers for any TRAP instructions which occur in your code or code emitted by the PASCAL compiler.
6. Set up the heap pointer, recover block pointer, open file list pointer, and any other pointers which are referenced implicitly by the compiler.
7. Decide where global variables will reside and set up register A5 to point to your global area. (Use the 'G' command when linking to establish the A5 relative starting point of your global data area.)
8. Set the status register for an appropriate mode and interrupt level.

9. Provide all utility routines called by the compiler:

```
integer and floating point math
string operations
set operations
heap management (to implement new, dispose, mark, release)
file operations
    open, close, read, write

many, many more
```

10. Read the BOOTROM 3.0 documentation to find out what environment is provided when control is first transferred to your system. Some useful information is retained in memory regarding physical resources which are present. The boot ROM also provides a few routines to do common operations which might be useful to some systems.

Rules for Using the Boot Command

A bootable disc is written using the Librarian's B command. (This is how the Boot disc for Pascal 2.0 is created.) The boot file **MUST** be created on a volume with a LIF directory, not on an SRM volume! After the disc is created, a copy of the system file can be moved out onto the SRM using the Filer.

There are two requirements for the boot command to work properly.

1. There must be no unresolved external references in any module transferred by the boot command. Each one must be a complete, standalone code segment. The Librarian will complain if you try to "B" a file with unresolved references.
2. All addresses and relocatable data must be resolveable at the address at which it is currently linked. The most common problem is 16 bit addresses that cannot reach their destinations. To get around these, you must use "patch jump space" during the linking process. See the Librarian documentation.

The address which serves as the load address is the current relocation base. For a standalone assembly module, this is specified with an ORG or RORG at the beginning of the code. For modules created or modified with the librarian, the relocation base is specified with the 'R' command while linking. Linking overrides the load addresses specified by RORG but not those specified by ORG.

Any compiled modules must be linked in order to specify the load address, since the relocation base emitted by the compiler is 0. Similarly, the 'R' command must be invoked when linking, since otherwise the librarian emits a relocation base of 0.

An Example

The following small program is provided as an example of using the Boot command. Follow the instructions below to create a boot disc.

1. Edit 'BTEST.TEXT', for either a 50 or 80 column screen.
2. Assemble 'BTEST.TEXT', producing 'BTEST.CODE'.
3. Execute the librarian.
4. Put an initialized LIF disk in #3.
5. Press 'B' for boot command.
6. Enter '#3:SYSTEM__X.' for name of system file on mini floppy.
7. Press 'I' for input file.
8. Enter 'BTEST' for the example code file.
9. Press 'T' or 'A' to transfer the file.
10. Press 'B' to finish "booting".
11. Cycle the power on your computer to load 'SYSTEM__X' and run it.

```
*
* EXAMPLE OF A MINIMAL BOOTABLE PROGRAM FOR 9826/9836
*
      rorg      -15000          default load address
*                               can be changed by linking

      start    *              beginning execution address

      lea      msg,a0          a0 indexes message

*      lea      $5121A0,a1      a1 indexes CRT memory, 80 cols
*      lea      $512704,a1      a1 indexes CRT memory, 50 cols

again  moveq   #0,d1           clear highlight byte
       move.b  (a0)+,d1        move message byte into register
       beq     *              infinite idle loop when finished
       move.w  d1,(a1)+        move highlight, character to screen
       bra     again          repeat

msg    dc.b    'Hello, this is a message.',0

       nosyms                  no symbol table dump, please
       end
```

Trap / Exception Vectors used in Pascal

These interpretations of the 68000 traps and exceptions are set up in the assembly language routine POWERUP, which stores values in the vector locations in high RAM. These values override the ones set up by the Boot ROM.

TRAPS

address	trap	escapecode	usual meaning
\$FFFFFF3A	15	(none)	debugger
\$FFFFFF40	14	-21	unassigned
\$FFFFFF46	13	-21	unassigned
\$FFFFFF4C	12	-21	unassigned
\$FFFFFF52	11	-21	unassigned
\$FFFFFF58	10	"N"	escape N
\$FFFFFF5E	9	(none)	non local goto
\$FFFFFF64	8	-3	dereference NIL pointer
\$FFFFFF6A	7	-8	value range error
\$FFFFFF70	6	-9	case statement error
\$FFFFFF76	5	-5	divide by zero
\$FFFFFF7C	4	-4	integer overflow
\$FFFFFF82	3	-3	I0result <> 0
\$FFFFFF88	2	-2	stack overflow
\$FFFFFF8E	1	(-2 if any)	link a6 emulator with stack check
\$FFFFFF94	0	(none)	PASCAL line breakpoint

SYSTEM EXCEPTIONS

address	escapecode	usual meaning
\$FFFFFFC4	-13	1111 opcode line
\$FFFFFFCA	-13	1010 opcode line
\$FFFFFFD0	(none)	TRACE
\$FFFFFFD6	-14	privilege violation
\$FFFFFFDC	-4	integer overflow (TRAPV)
\$FFFFFFE2	-8	CHK instruction
\$FFFFFFE8	-5	divide by zero (hardware)
\$FFFFFFEE	-13	illegal instruction
\$FFFFFFF4	-11	address error
\$FFFFFFFA	-12	bus error

Chapter 19

Device I/O

Introduction

This discussion of I/O is on 'device I/O'. In general the device I/O facilities deal with the interface cards that plug into the computer backplane or appear to be logically plugged into the backplane. To better understand this device I/O discussion, familiarity with the I/O documentation (in the Pascal Procedure Library User's Manual) is very strongly recommended.

The goal of device I/O is to provide the I/O facilities of the computer hardware in a general way to both the Pascal language system and the Pascal programmer. Device I/O is used by the programmer via the system library facility. Device I/O is used by the system to access external printers, disks, et cetera via various low level device drivers.

The Hardware View

The physical memory map of the computer is defined as:

----- \$FFFFFF \$800000 -----	RAM
\$7FFFFFF \$600000 -----	External I/O
\$5FFFFFF \$400000 -----	Internal I/O
\$3FFFFFF \$000000 -----	ROM - system and BOOT ROM

The internal interfaces are in the \$400000 through \$5FFFFFF range. These interfaces include the built in disk drives, keyboard, and CRT devices. This area also includes the built in HP-IB interface and the plug in DMA interface (HP 98620A). Although the actual locations vary the intent is that these interfaces will occur on 65536 (\$FFFF) boundaries.

The external interfaces (i.e. HP 98624A, et cetera) exist in the \$600000 through \$7FFFFFF address range. These interfaces occur on 65536 (\$FFFF) boundaries. As of this writing, only the HP 98627 color graphics card violates this by using 128K bytes in the external I/O address space.

Note that, even though the built in HP-IB interface is in the internal I/O address space, it is handled via the device I/O system. This is due to the fact that the interface is almost identical to the external HP-IB interface (HP 98624A). Handling the internal HP-IB interface any other way would cause unnecessary overhead.

The 9816 computer has a built in RS-232 interface. This interface is at address \$690000 and so is an 'external' interface and requires no special handling. This interface is equivalent to the 98626 interface card.

The Programmer View

The intent of the Pascal device I/O system is to present a logical view of the I/O hardware that is very similar to the BASIC language view of the I/O hardware. BASIC defines, fundamentally, 31 different select codes. These select codes and their definition are:

Select code	device	address/type
-----	-----	-----
1	CRT display	internal
2	keyboard	internal
3	graphics	internal
4	floppy	internal
5	power fail option	internal
6	reserved	
7	built-in HP-IB	internal \$478000
8	plug in interface	external \$680000
9	plug in interface	external \$690000
10	plug in interface	external \$6A0000
11	plug in interface	external \$6B0000
.	.	.
.	.	.
30	plug in interface	external \$7E0000
31	plug in interface	external \$7F0000

The Pascal system is not exactly the same. Its definition for the programmer view of the I/O select codes is:

Select code	device	address/type
-----	-----	-----
1	CRT display	internal
2	keyboard	internal
3	dma	internal
4	unused	
5	unused	
6	unused	
7	built-in HP-IB	internal \$478000
8	plug in interface	external \$680000
9	plug in interface	external \$690000
10	plug in interface	external \$6A0000
11	plug in interface	external \$6B0000
.	.	.
.	.	.
30	plug in interface	external \$7E0000
31	plug in interface	external \$7F0000

Note that in both of these schemes there are only 31 logical select codes presented to the user. The hardware supports 64 different select codes (32 internal and 32 external). Further note that there are gaps in what external cards can be directly accessed -- any external (plug in) interface set to select code 0 through 7 will not be recognized by the Pascal I/O system.

Earlier it was mentioned that device I/O is not concerned with the built in CRT or keyboard. These internal devices are included as a convenience to the programmer (for debugging). The device drivers use the standard Pascal READ and WRITE routines. BASIC has the keyboard and CRT on two separate select codes and allows writes to the keyboard select code for various reasons. The Pascal system views select codes 1 and 2 as identical -- a write is sent to the CRT and a read gets data from the keyboard. Also -- input, output, and reset operations are all that are supported by Pascal for select codes 1 and 2.

In general, the normal programmer will deal with these device I/O facilities through the system library I/O modules. The system programmer use (such as the internal HP use of this device I/O) is done at a low, driver level for the most part. This is done so that the system does not require major pieces of the I/O library to be resident for normal operations. Applications programs should use the library modules.

The user's view of the library is of a collection of modules. The modules are:

Level	Use	GENERAL	HPIB	SERIAL
0	status/control	GENERAL_0	HPIB_0	SERIAL_0
1	simple I/O	GENERAL_1	HPIB_1	-----
2	enter/output	GENERAL_2	HPIB_2	-----
3	status/control	GENERAL_3	HPIB_3	SERIAL_3
4	transfer	GENERAL_4	-----	-----

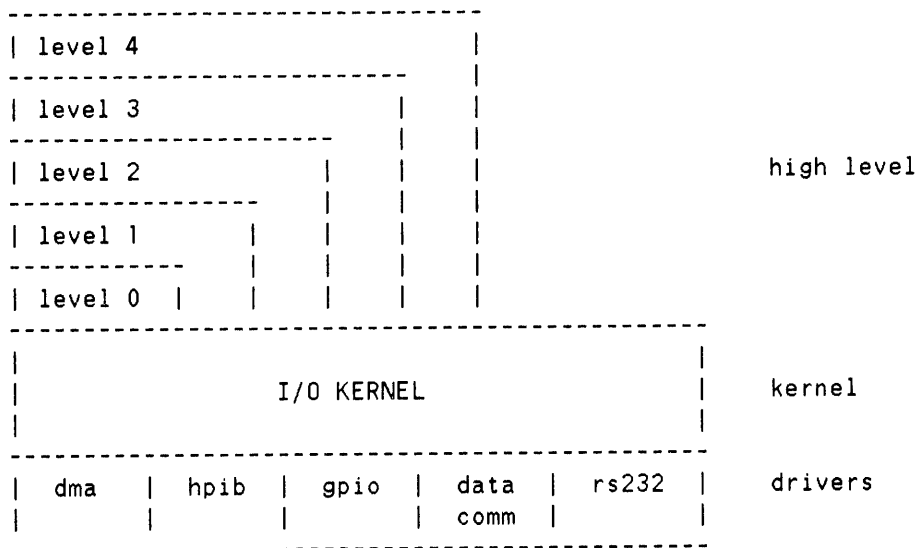
In addition to the these modules there are two other modules of interest to the user: IODECLARATIONS and IOCOMASM. IODECLARATIONS contains the library's constant, type, and variable declarations. IOCOMASM is a support module that contains binary operations (like binand etc.).

General Architecture

In the following device I/O discussion, the term kernel will mean the device I/O kernel -- not the Pascal system kernel. Any reference to the Pascal kernel will be specifically mentioned as such. The I/O Library consists of three primary collections of modules:

1. Kernel modules
2. Driver modules
3. IO library high level modules

The kernel modules are the base on which the rest of the device I/O system depends. It allocates and initializes the various tables used by the underlying drivers and the high level routines. The drivers are independent entities that can, for the most part, be added or deleted at will. These low level drivers do not depend on other driver modules. They do, however, depend on some of the kernel facilities. The high level routines are a set of layered modules that reside on top of the kernel structure.



These collections of modules reside in three places in the released system -- INITLIB on the BOOT volume, IO on the LIB volume and in various files on the CONFIG volume. When you configure your system to use the I/O facilities there will only two primary places for these modules -- INITLIB on the BOOT volume and LIBRARY on the SYSVOL volume. The normal user documentation explains how to configure your system appropriately.

The kernel code and the installed driver code reside in INITLIB. The kernel export text, IO library code, and IO library export text (originally in file IO on the LIB volume) are installed in the file LIBRARY on the SYSVOL volume. It is possible for a user to extend the drivers and/or the high level routines and place these extensions in other files in the system. This approach however, will not allow the facilities to be used automatically. It is also possible to not install the drivers in the INITLIB file on the BOOT volume but load the drivers by executing the appropriate file on the CONFIG disk. Similarly, it is possible to not install the I/O code into the file LIBRARY on the SYSVOL volume. This would require that your source code do a \$SEARCH\$ prior to any IMPORTs of I/O routines. This is not recommended. The remainder of the discussion on I/O will assume that the I/O system is installed in a normal fashion.

The kernel modules consist of the following:

1. IODECLARATIONS
2. IOCOMASM
3. GENERAL__0
4. IOLIBRARY__KERNEL

These four kernel modules are linked together under the name IODECLARATIONS. Each INITLIB module can also have an executable program segment that gets executed at the time it is loaded. IOLIBRARY__KERNEL is an executable program (the other three are not) and it allocates and initializes the static read/write memory used by the rest of the device I/O system. This program also allocates the temporary storage for any card that exists, independent of whether there is or is not a driver for it. The IOLIBRARY__KERNEL program consists of a single call to a GENERAL__0 routine, KERNEL__INITIALIZE, which does all of the initialization work. This initialization routine MUST execute only once -- at INITLIB load/execute time.

The driver modules consist of the actual assembly or PASCAL routines that deal with a specific interface card. There is also an executable program segment for each driver module. This program searches the select code table in the static r/w initialized by the KERNEL general__0 module for all select codes that have the right interface card (HPIB drivers will search for the 98624 interface). This program will then set up the driver tables to point to the correct drivers.

The rest of the IOLIB modules are high-level modules that are used by an end user in his/her application program. These routines make calls to the low-level drivers.

From the low level drivers view, IOCOMASM not only contains binary operations for the user but also contains utilities used by the assembly language driver code (HPIB and GPIO in particular).

The IODECLARATIONS module and some set of driver modules will exist in the INITLIB file as object code (not EXPORT text). The export text will reside on the LIBRARY file. The rest of the library will reside in the LIBRARY (export text and code). Note that the low level drivers do not have any export text anywhere in the released system.

To put the modules into perspective -- the following table contains all the identifiable modules that exist in the device I/O system, where they came from, and how big they are. Each of the modules has two main size attributes -- global space and code size. The following table shows the approximate size of each of the modules.

module name	code size	global size	code location	export text	source file (P)=PASCAL (A)=ASSEMBLY
IODECLARATIONS	3798	1082	INITLIB	(N) (N)	KERNEL (P) COMASM (A)
HPIB	3846	120	INITLIB	(N) (N)	HPIB (A) H_DRV (P)
GPIO	2174	120	INITLIB *	(N) (N)	GPIO (A) G_DRV (P)
DATA_COMM	5196	126	INITLIB *	(N) (N)	DC (A) DC_DRV (P)
DMA	506	0	INITLIB	(N)	DMA_DRV (P)
RS232	3132	120	INITLIB *	(N) (N)	RS_DRV (P) RS (A)
IODECLARATIONS	0	0	LIBRARY **	(Y)	KERNEL (P)
IOCOMASM	0	0	LIBRARY **	(Y)	COMASM (A)
KERNEL_INITIALIZE	0	0	LIBRARY **	(Y)	KERNEL (P)
GENERAL_0	0	0	LIBRARY **	(Y)	KERNEL (P)
GENERAL_1	650	0	LIBRARY **	(Y)	IOLIB (P)
GENERAL_2	1972	0	LIBRARY **	(Y)	IOLIB (P)
GENERAL_3	1982	0	LIBRARY **	(Y)	IOLIB (P)
GENERAL_4	2238	0	LIBRARY **	(Y)	IOLIB (P)
HPIB_0	210	0	LIBRARY **	(Y)	IOLIB (P)
HPIB_1	1854	0	LIBRARY **	(Y)	IOLIB (P)
HPIB_2	1038	0	LIBRARY **	(Y)	IOLIB (P)
HPIB_3	604	0	LIBRARY **	(Y)	IOLIB (P)
SERIAL_0	1592	0	LIBRARY **	(Y)	IOLIB (P)
SERIAL_3	2802	0	LIBRARY **	(Y)	IOLIB (P)

* - This module might be in INITLIB or on the CONFIG disk.

** - This module might be in LIBRARY or in IO on the LIB disk.

The modules IODECLARATIONS, IOCOMASM and KERNEL_INITIALIZE exist in the LIBRARY as code-less entities for the express purpose of providing user-accessible export text.

Main Data Structures

The way the the kernel supports the underlying drivers and the high level routines is primarily through a set of common data structures. These data structures consist of 3 primary structures:

1. A select code table -- ISC__TABLE
2. Driver R/W
3. Buffer Control Blocks

The select code table is intended primarily for use by the high level routines and directly by the Pascal (and system) programmer. The driver R/W is intended for use by the low level driver code. These two structures are the interface between the high or low level code and the kernel.

The select code table is not currently used by the low level drivers (and you will see duplication of information between the driver R/W and the ISC__TABLE because of this fact). The driver R/W has some fields intended for use by the higher level routines. This high level use of the driver R/W was done to reduce the size requirements of the select code table -- every possible select code is allocated an entry in the select code table, not every select code has driver R/W.

The buffer control blocks are a specialized interface mechanism between the high and low level code specifically for buffered transfers.

ISC__TABLE

As mentioned previously, the high level routines utilize a select code table. The INITLIB kernel allocates and initializes this table. This table needs to contain all the information necessary to support the high level use of the low level drivers. To support this use, there are four primary pieces of information (and some secondary information) that the high level routines need, which are:

1. Where are the low level drivers.
2. Where is the driver R/W for this select code.
3. What sort of interface is this.
4. Where is the interface card.

The table to support these needs is defined as:

```
VAR isc_table : PACKED ARRAY [ 0..31 ] OF isc_table_type;

TYPE isc_table_type =
  RECORD
    io_drv_ptr: ^drv_table_type; { where are the drivers }
    io_tmp_ptr: pio_tmp_ptr;     { where is the drv R/W }
    card_type : type_of_card;    { what sort of card   }
    user_time : INTEGER;         { user timeout       }
    card_id   : type_card_id;    { what sort of card   }
    card_ptr  : ANYPTR;          { where is the card   }
  END;
```

ISC_TABLE is a static, global array with indices of 0 to 31. 0 is reserved. Select codes 1 through 7 are pre-defined internal interfaces. Select codes 8 through 31 are defined to be external interfaces.

Each of the entries in ISC_TABLE has the following meaning:

io_drv_ptr: Contains a pointer to a table of low level drivers.

io_tmp_ptr: Contains a pointer to the driver R/W area (permanent heap) used by the low level drivers and system routines.

card_type: Contains a short integer signifying the generic interface type (such as HPIB or SERIAL). The integer was chosen over enumerated type for extensibility of card types. The defined card type constants are:

```
no_card      = 0 ;
other_card   = 1 ;
system_card  = 2 ;
hpib_card    = 3 ;
gpio_card    = 4 ;
serial_card  = 5 ;
graphics_card = 6 ;
srm_card     = 7 ;
```

If you (the customer) need a new type, use negative numbers. Positive numbers are reserved for Hewlett Packard use.

user_time: Contains a 32 bit integer containing the timeout period in milliseconds.

card_id: Contains a short integer signifying the specific interface type (such as HP 98626). As with card type, integers were chosen for extensibility. Positive values for card id indicate that the id value is the actual card hardware id bits (in the hardware register # 1 |_ bits # 0 .. 4). Negative id values indicate a 'pseudo-id'. The defined constants for card id are:

```
hp98628_dsnd1 = -7;          { DSN/DL          }
hp98629       = -6;          { resource manager }
hp_datacomm   = -5;
hp98620       = -4;          { dma              }
internal_kbd  = -3;
internal_crt  = -2;
internal_hpib = -1;
no_id        = 0;
hp98624       = 1;           { hpib             }
hp98626       = 2;           { serial           }
hp98622       = 3;           { gpio             }
hp98623       = 4;           { bcd              }
```

```

hp98625      = 8;          { disk          }
hp98628_async = 20;       { async version }
hp98627      = 28;       { graphics     }

```

If you (the customer) need a new id, use negative numbers of -1024 and beyond (-1025, -1026, etc.). Note that there is no guarantee of uniqueness for user generated card_id's.

card_ptr: Contains an address specifying the hardware address of the interface card (like \$478000 for the internal HP-IB interface).

Card_ptr and io_temp_ptr should have unique values for each card. The driver pointer points to a table of low level routines that are shared between a common card type. For example -- there might be an HP-IB interface plugged in plus the internal HP-IB. The table entries for these 2 interfaces might look like -

```

isc_table[7]      card_id      := internal_hpib
                  card_type    := hpib_card
                  io_drv_ptr   := ANYPTR(hpib_drivers)
                  io_temp_ptr  := temp_space_x
                  card_ptr     := HEX 478000

isc_table[8]      card_id      := hp98624
                  card_type    := hpib_card
                  io_drv_ptr   := ANYPTR(hpib_drivers)
                  io_temp_ptr  := temp_space_y
                  card_ptr     := HEX 680000

```

External cards can have the select code switches set to 0 through 7. These cards will not be found by the system. It is possible to fool the system into finding them by setting the card_ptr field in ISC_TABLE[x] and the card_addr field in ISC_TABLE[x].IO_TMP_PTR^ to the address of the card you wish to deal with (where x is an unused select code in ISC_TABLE -- like select codes 4,5,6 or one of the unused in the 8 through 31 range). Note that you will have to take care of setting up the remaining fields in the various data structures, linking in an ISR (if appropriate) and resetting the interface. Note that for most applications, the 24 external available select codes should sufficient.

Driver Read/Write

When the kernel module IODECLARATIONS is brought in at INITLIB time, the program (and module) KERNEL_INITIALIZE is executed. This code allocates a block of R/W memory (heap space) for each I/O card plugged into the back plane. This memory is allocated with the system procedure NEW. To insure that the space does not go away, the system marks a new start of heap after all INITLIB modules have been executed (described elsewhere). This area looks like:

```
PACKED RECORD
  myisrib   : ISRIB ;           { system ISR linkage   }
  user_isr  : io_funny_proc;    { user ISR linage    }
  user_parm : ANYPTR ;
  card_addr : ANYPTR ;         { card location      }
  in_bufptr : ANYPTR ;         { ptr to buf ctl block }
  out_bufptr : ANYPTR ;        { ptr to buf ctl block }
  eirbyte   : CHAR ;
  my_isc    : io_byte ;        { select code        }
  timeout   : INTEGER ;       { driver timeout     }
  addressed : io_word ;       { bus address, etc.  }
  drv_misc  : ARRAY[1..xxx] OF CHAR ; { actual driver R/W  }
END;
```

Each of the entries has the following meaning:

myisrib: A block of information (called an ISRIB |_ Interrupt Service Routine Information Block) used by the operating system to set up interrupts (described elsewhere).

user_isr: A procedure variable that contains a procedure to be called by the low level drivers if a user specified interrupt condition occurred. This is not THE driver ISR, it is just a procedure that gets called AFTER the driver ISR is done (if appropriate). The procedure is of type io_funny_proc |_ this type looks like:

```
TYPE io_funny_proc =
  RECORD
    CASE BOOLEAN OF
      TRUE:
        (real_proc : io_proc);
      FALSE:
        (dummy_sl   : ANYPTR ; { static link }
         dummy_pr   : ANYPTR) { proc addr   }
    END;
```

The type io_proc is defined as:

```
TYPE io_proc : PROCEDURE (temp : ANYPTR);
```

What all this boils down to is the user_isr procedure is a procedure with a single ANYPTR parameter. The io_funny_proc record is necessary so that the I/O

system can set up NO procedure (procedure address and static link are NIL) at initialization times.

user_parm: The user isr procedure receives this parameter when the procedure is called.

card_addr: This field contains a pointer to the interface hardware address |_ identical to ISC_TABLE[x].card_ptr. This is used by the low level drivers because they do not have access to ISC_TABLE.

in_bufptr: This field contains a pointer to a buffer control block IF there is an active inbound transfer between the buffer control block's buffer and this select code.

out_bufptr: This field contains a pointer to a buffer control block IF there is an active outbound transfer between the buffer control block's buffer and this select code.

eirbyte: This field is a byte used by some of the drivers for an enable interrupt facility |_ state information.

my_isc: This field contains a byte with the value of the select code that this interface is specified to be at (a sort of back pointer to the ISC_TABLE). This field is used by the IO_FIND_ISC(tmp_ptr) function.

timeout: A copy of user_time in ISC_TABLE timeout value. This is in two places because the system printer and disk drivers have their own timeouts. Whenever a shared interface (like HP-IB) goes through re-addressing the user_time value will get copied into timeout.

addressed: This field is a short integer and indicates whether the interface is a shared (or bus) interface. A -1 indicates that it is not shared. A positive value indicates the actual device bus address.

drv_misc: This field is an array of characters that is to be used by the drivers as they see fit. This area is the 'actual' driver R/W. This area comes in three sizes |_ 32, 128, and 180 characters. Unrecognized interfaces are given 180 characters. If a new card is given the 180 character field, the driver can throw the temp space that the kernel gave the drivers and allocate a new block (as long as it is appropriately initialized).

Buffer Control Block

The buffer control block is a user created structure that contains all the information necessary for control of a buffer transfer and a pointer to a buffer. Normally the block is statically created by a user VAR. At the static allocation time, there is no valid information in the block and there is no allocated buffer space. When the user calls IOBUFFER(block,size), a data area is created on the heap (with NEW) and the block is initialized to a clean, valid state. When a transfer starts, the control block is bound to a select code for the duration of the transfer. The buffer control block looks like:

```
RECORD
  drv_tmp_ptr : pio_tmp_ptr;           { ptr to driver R/W   }
  active_isc  : io_byte;               { select code         }
  act_tfr     : actual_tfr_type ;      { given transfer mode }
  usr_tfr     : user_tfr_type ;        { requested tfr mode  }
  b_w_mode    : BOOLEAN ;              { byte/word mode     }
  end_mode    : BOOLEAN ;              { end/eoi mode       }
  direction   : dir_of_tfr ;           { direction of tfr   }
  term_char   : -1..255 ;               { termination character }
  term_count  : INTEGER ;              { transfer count     }
  buf_ptr     : ^buf_type ;            { ptr to buffer area }
  buf_size    : INTEGER ;              { maximum buf size   }
  buf_empty   : ANYPTR ;               { next datum in buffer is }
  buf_fill    : ANYPTR ;               { where to put next datum }
  eot_proc    : io_funny_proc;         { end of transfer link }
  eot_parm    : ANYPTR ;
  dma_priority: BOOLEAN;
```

END;

drv_tmp_ptr: This field contains a pointer to the driver R/W when a transfer is in progress. The temp space also contains a pointer to the buffer control block when a transfer is active.

active_isc: This byte field contains the select code currently active with this buffer. When no transfer is active the field contains a 255 (= constant NO_ISC).

act_tfr: This field is an enumerated type that indicates what type of transfer the driver is using (as opposed to what the user requested). The values are:

```
no_tfr
INTR_tfr
DMA_tfr
BURST_tfr
FHS_tfr
```

usr_tfr: This field is an enumerated type that indicates what type of transfer the user requested (as opposed to what the driver gave the user). The values are:

```
dummy_tfr_l           overlap_INTR
serial_DMA            overlap_DMA
serial_FHS            overlap_FHS
```

	serial_FASTEST	overlap_FASTEST
	dummy_tfr_2	OVERLAP

b_w_mode: This is a boolean field that contains an indication of whether or not the transfer is a byte or word mode transfer.

end_mode: This is a boolean field that contains an indication of whether or not the transfer is to have an EOI/END termination (input) or is to send an EOI/END (output).

direction: This field contains an enumerated type indicating transfer direction (to_memory or from_memory).

term_char: This short integer field contains a -1 if there is no character termination. A 0..255 value indicates that the specified character value is the termination character.

term_count: This integer field holds the maximum transfer count.

buf_ptr: This field contains a pointer the actual buffer space. The buffer is viewed as a large packed array of characters.

buf_size: This integer field holds the maximum buffer size.

buf_empty: This field contains a pointer to the first valid character in the buffer. This pointer is incremented when a character is read from the buffer.

buf_fill: This field contains a pointer to the first empty character position in the buffer. This pointer is incremented when a character is put into the buffer.

eot_proc: This field is a procedure variable. This procedure is called when a transfer finishes. This procedure follows the form of user_isr in io_tmp_ptr^.

eot_parm: This paramter is passed to the user's eot procedure.

dma_priority: This boolean field if true indicates a DMA transfer is to be given high hardware priority. The DMA hardware has two channels. Basically the priority indicates whether or not the DMA hardware will allow bus cycles by the CPU between very fast requests. This is only required if the transfer rate is greater than 300K transfers/second.

Driver Structure

As mentioned earlier, the kernel is the base for the various drivers and high level routines. The kernel is presented primarily through the `ISC_TABLE` structure. The other main mechanism for supporting device I/O is a common structure for the routines that comprise the device driver. This section discusses the form of a common driver structure.

The goal of the device I/O system is similar to the goal of the underlying file system structure -- extensibility. The intent is that at a later date HP (or a customer) could extend the I/O system and add new drivers and still have major portions of the system work with the new interface.

The major aspect of this extensibility is a common set of 'atomic' operations that all interface drivers support. Fundamentally, a driver consists of two pieces: the low level drivers and an initialization program. The low level drivers are contained in a table of procedure variables (this is what is in the 120 bytes of global space in the driver modules `HPIB` et cetera -- the driver table variable). The table consists of the following set of procedures:

```
RECORD
  iod_init   : io_proc ;      { initialization   }
  iod_isr    : ISRPROCTYPE ;  { interrupt routine }
  iod_rdb    : io_proc_vc ;   { read a character  }
  iod_wtb    : io_proc_c ;    { write a character }
  iod_rdw    : io_proc_vw ;   { read a word       }
  iod_wtw    : io_proc_w ;    { write a word      }
  iod_rds    : io_proc_vs ;   { read status       }
  iod_wtc    : io_proc_s ;    { write control     }
  iod_end    : io_proc_vb ;   { end(eoi) test    }
  iod_tfr    : io_proc_ptr ;  { transfer          }
  iod_send   : io_proc_c ;    { send ATN msg     }
  iod_ppoll  : io_proc_vc ;   { parallel poll     }
  iod_set    : io_proc_l ;    { set interface line }
  iod_clr    : io_proc_l ;    { clr interface line }
  iod_test   : io_proc_vl ;   { tst interface line }
END;
```

The procedure variable types used above are defined as:

```
TYPE
  io_proc      = PROCEDURE (temp : ANYPTR);
  io_proc_c    = PROCEDURE (temp : ANYPTR ; v      : CHAR  );
  io_proc_vc   = PROCEDURE (temp : ANYPTR ; VAR v : CHAR  );
  io_proc_w    = PROCEDURE (temp : ANYPTR ; v      : io_word);
  io_proc_vw   = PROCEDURE (temp : ANYPTR ; VAR v : io_word);
  io_proc_s    = PROCEDURE (temp : ANYPTR ; reg    : io_word ;
                           v      : io_word);
  io_proc_vs   = PROCEDURE (temp : ANYPTR ; reg    : io_word ;
                           VAR v : io_word);
  io_proc_l    = PROCEDURE (temp : ANYPTR ; line   : io_bit );
  io_proc_vl   = PROCEDURE (temp : ANYPTR ; line   : io_bit ;
                           VAR v : BOOLEAN);
  io_proc_vb   = PROCEDURE (temp : ANYPTR ; VAR v : BOOLEAN);
  io_proc_ptr  = PROCEDURE (temp : ANYPTR ; v      : ANYPTR );
```

This set of procedures was decided upon because they were "atomic" for the desired operations within the Pascal system. In the initial investigation BASIC drivers were going to be used but BASIC was tuned specifically to the BASIC language and the BASIC line and end of line structure (including line temporaries). This overhead would have been too great for the Pascal system. The code and structure that was used for the low level drivers (for hpib and gpio) was based on the 9826 HPL code. Its structure closely matched what was needed. Note that the code is NOT IDENTICAL. Primary differences occur in the ISR's, transfers, and the way that the drivers are connect to the rest of the language system.

In looking at the set of driver procedures, not all of them are obviously 'atomic'. What is 'atomic' depends on your needs. The general uses and needs for the various procedures are:

- iod_init: This procedure is called whenever IORESET(sc) is called, whenever IO_INITIALIZE or IO_UNINITIALIZE is called, whenever the STOP or CLR I/O keys are pressed, and at system load.
- iod_isr: This procedure is called whenever the specified interface generates an interrupt.
- iod_rdb: This procedure is called from the various procedures in modules GENERAL_1 and GENERAL_2 that input data.
- iod_wtb: This procedure is called from the various procedures in modules GENERAL_1 and GENERAL_2 that output data.
- iod_rdw: This procedure is called from the READWORD function in GENERAL_1.
- iod_wtw: This procedure is called from the WRITEWORD procedure in GENERAL_1.
- iod_rds: The read status routine is called by IOSTATUS and by some of the interface specific modules |_ especially HPIB_1, HPIB_2, HPIB_3, SERIAL_0 and SERIAL_3. Any library use of status is preceded by a interface id check.
- iod_wtc: The write control routine is called by IOCONTROL and by some of the interface specific modules |_ especially HPIB_1, HPIB_2, HPIB_3, SERIAL_0 and SERIAL_3. Any library use of control is preceded by a interface id check.
- iod_end: This procedure indicates whether EOI (END) was set on the last byte read. This is used by the END_SET function in module HPIB_1. It is also used by the various disk drivers.
- iod_tfr: The transfer procedure is the low level code that handles the buffer transfers. It is called from various GENERAL_4 procedures and by the various disk drivers.

iod_send: The send command procedure sends a bus command on an HP-IB interface. It is used in various places in the library and disk drivers. All uses are preceded by a check for an HP-IB interface.

iod_ppoll: The ppoll procedure performs a parallel poll on an HP-IB interface. It is used in the PPOLL function in module HPIB_3 and in the disk drivers. All uses are preceded by a check for an HP-IB interface.

iod_set,iod_clr,iod_test:
 These procedures allow for checking and setting the interface lines. Only HP-IB and GPIO have implemented these procedures. Data comm and the RS-232 drivers use status and control to implement these features. The only library call is in HPIB_0 and is preceded by an HPIB card test.

At an absolute minimum in most applications only three procedures in this table are required -- read a character, write a character and initialize. What else is needed depends on what your drivers need to do.

The main additions to the 'true' atomic operations are the HPIB specific procedures. These were added so that the disk drivers could be implemented with only the low level code. The procedures are just what is needed to implement most disk driver functions.

Note that all the procedures have only one parameter in common -- io_tmp_ptr. This is necessary so that the code will operate on the appropriate r/w space. The card address is also necessary but it is contained in the r/w space so the low level drivers do the look up there -- rather than take the time to pass it in as a parameter.

These procedures are all PROCEDURES -- none of them are FUNCTIONS. This is due to the manner in which they are called. The drivers, because they are indexed out of a table, must be called with the CALL statement in Pascal. The calling facility does not provide any mechanism for returning values. So all values returned are done by a VAR parameter in the calling parameter list. An example of the calling of the drivers looks like:

```
CALL(isc_table[mysc].io_drv_ptr^.iod_rdb ,
     isc_table[mysc].io_tmp_ptr^ ,
     mychar);

CALL(isc_table[mysc].io_drv_ptr^.iod_wtb ,
     isc_table[mysc].io_tmp_ptr^ ,
     'x');
```

High Level Routines

The high level routines of the library are fairly straight forward. The higher numbered modules contain more powerful facilities.

One of the major aspects of the high level routines that needs some explanation is the select code / device parameter. Most of the general purpose routines allow either a select code (i.e. 7) or a device specifier (i.e. 705). The way that this works is with a trick of organization. All the modules that allow both select codes and devices call some addressing routines in the module `HPIB__1`. These routines are functions that return a select code. The calling routines pass in the select code / device and the addressing routines perform the appropriate addressing (and then return a select code to the calling routine).

From an organizational (and structured) point of view, it would be better to relegate the addressing features to `HP-IB` modules only. This would make the library more difficult to use by the programmer.

The addressing routines consist of four procedures:

<code>ADDR_TO_LISTEN</code>	<code>SET_TO_LISTEN</code>
<code>ADDR_TO_TALK</code>	<code>SET_TO_TALK</code>

The two sets are different in that the `ADDR_TO__` routines are intended for use by data transfer routines (like `READNUMBER` and others in `GENERAL__2` and `GENERAL__4`) whereas the `SET_TO__` are intended for use by bus control routines (like `TRIGGER` and others in `HPIB__2` and `HPIB__3`). The `ADDR_TO__` will wait to be addressed if the `HP-IB` card is not the active controller. The `SET_TO__` routines generate an error if the `HP-IB` card is not the active controller.

Execution Walkthrough

This section is included to help tie things together. It will show various steps in the execution of the device I/O system. The steps included in this walkthrough include power-up, stop key, program compilation and program execution. As a suggestion, read through the section first and then, using the system listings, go through the steps and look at the actual code.

Power-Up

For the device I/O system, the first time of interest is at INITLIB load time. The kernel and drivers reside as linked modules in INITLIB on the BOOT device (mini-floppy or shared resource manager). As each module is brought in, it is executed if it has an execution address. The order of device I/O modules in INITLIB is IODECLARATIONS followed by any device drivers (i.e. INIT_HPIB et cetera). The IODECLARATIONS module contains the kernel modules (iodeclarations, iocomasm, general__0 and iolibrary__kernel).

After IODECLARATIONS is brought in, it is allocated its global space (including the select code table) and then it is executed. This execution goes through and searches for the various interfaces (both internal and external) that the device I/O system is concerned with. As it finds the interfaces, it initializes the ISC_TABLE entry with appropriate values. If the interface requires driver R/W (or if the kernel doesn't know for sure), heap space is allocated and initialized. The driver field of the ISC_TABLE for every select code is set to point to a set of dummy drivers (that generate an error if ever called). Then select codes 1 and 2 are set up with their simple drivers. The last thing that the kernel does is to set a CLRIOHOOK. This hook is a system hook that is called when the STOP or CLR I/O keys are pressed. It is also called when the system catches an escape from any program. This hook is set to the routine IO_SYSTEM_RESET which is the same routine called by IOINITIALIZE and IOUNINITIALIZE.

Now that the kernel has done its duty, drivers can start being loaded. For this walkthrough the HPIB driver will be discussed. After INIT_HPIB (which contains exth, hpib__initialize, and init__hpib) is loaded, it is allocated its globals. These globals consist of 120 bytes for its driver table (the entity that the driver pointer field in ISC_TABLE points to). When execution starts, it will first set up this driver table with procedure values taken from the exth driver module. The approach taken for this is to first set the drivers up as dummy drivers (all errors if ever called) and then fill in the valid routines -- i.e. iod__wtb := eh__wtb; -- this insures all entries in the driver table are valid (if errors).

The hpib driver then searches for all valid HP-IB interfaces and sets the ISC_TABLE driver pointer to the hpib driver table. It then links in the system interrupt for the interface. When this has been done for all interfaces, the interfaces are reset, one by one. Note that at the start of the hpib initialization code there is a string (IO_REVID). This string is intended to be used by the HP field service force to determine which revision of the drivers you have installed in your machine.

Stop Key

When the STOP or CLR I/O key is pressed, the CLRIOHOOK calls the IO_SYSTEM_RESET routine in module GENERAL__0. This routine goes through each select code and sets the user timeout value to zero (infinite timeout). It also sets the driver R/W entries that it understands to valid defaults. For example, user isr's are set to 'no isr' and driver timeouts are set to zero (infinite). When these values are set for all select codes, the IO_SYSTEM_RESET routine then goes through and calls the reset driver for each interface. Finally, to insure availability of the DMA resources, the DMA channels are released.

Program Compilation and Execution

To show what happens in a typical compilation/execution sequence, a sample program will be used. The program is:

```
PROGRAM TEST(INPUT,OUTPUT);
IMPORT GENERAL_2;
BEGIN
  WRITENUMBERLN(701 , 23.45);
END.
```

When this program text file is compiled, the compiler (after encountering the 'IMPORT GENERAL__2;' line) will search the system library for module GENERAL__2. Having found that module, it will include the export text in GENERAL__2 into the program (without listing it) as if the user had it in his/her text file. This phase is recursive in that GENERAL__2 imports other I/O library modules that in turn import system modules. This is why you see so many dots on the screen when you import I/O modules. Note that if you have HP-IB or Shared Resource Manager mass storage or if you get a listing then the I/O system is called by the file system to bring in the compiler, library code and text files, to put out the code files and to generate the listing.

When the program code file is executed, the first step is to load the code file. This load will result in unresolved external references. To see these references, try to load the program code file with the SYSVOL (or whatever volume contains LIBRARY) turned off. When the code file is loaded, the system library is searched for the unresolved references. For this sample program the modules are GENERAL__2, HPIB__1 and IODECLARATIONS. Note that IODECLARATIONS already is in the machine because of the kernel. So, GENERAL__2 and HPIB__1 are loaded.

When the program starts actual execution, note that no I/O reset was performed by the system or by the user. The drivers are, however in a good state because the drivers themselves did a reset at INITLIB load/execution time.

The WRITENUMBERLN routine is called with 701 as the device parameter and 23.45 as the real number to be written. WRITENUMBERLN then does an ADDR__TO__LISTEN with address 701.

ADDR__TO__LISTEN breaks out select code and device address (7 and 1). ADDR__TO__LISTEN then checks to see if the interface is addressable (since the operation is to an addressed device within the select code). Since it is addressable, ADDR__TO__LISTEN then sends the 3 HP-IB commands for MY TALK, UNLISTEN, and DEVICE 1 LISTEN. These commands are sent via SEND_COMMAND. SEND_COMMAND calls the low level driver by doing a 'CALL ISC_TABLE[isc].io_drv_ptr^.iod_send (temps , command);'.

The `iod_send` routine in the HP-IB drivers is the `eh_send` routine. This assembly language routine checks for an active transfer (so you don't mess up an overlap transfer). It then sets the attention line (which checks for active control). Then the `iod_send` routine checks what sort of command was being sent so that it can set up the TI 9914 interface chip appropriately.

`ADDR_TO_LISTEN` finally returns back just the select code (7) to the `WRITENUMBERLN` routine. `WRITENUMBERLN` then calls `WRITENUMBER` with a select code of 7 and a real number of 23.45.

`WRITENUMBER` uses the system number formatter via the `STRWRITE` routine. `STRWRITE` works like `WRITE(xxxxx)` except that the destination of the write is a string variable instead of the screen. So, `STRWRITE` puts the characters '23.45' into the string `IO_WORK_STRING`. `WRITENUMBER` then does a `FOR` on the string and sends each byte of the string to the select code via the `iod_wtb` routine.

The `iod_wtb` routine in the HP-IB drivers is the `eh_wtb` routine. This assembly language routine checks that the HP-IB interface is addressed as a talker. It then makes sure that the attention (command) line is false. It then makes sure that the interface is ready for the next byte. It then can put the character into the interface.

When all the characters have been sent by `iod_wtb` inside of `WRITENUMBER`, `WRITENUMBER` returns to `WRITENUMBERLN`. `WRITENUMBERLN` then calls the `iod_wtb` routines twice with a carriage return and a line feed character.

`WRITENUMBERLN` is now finished and returns to the main program. The program, itself, is finished and terminates with no further I/O activity.

Low Level Drivers

HP-IB

The HP-IB low level drivers consist of three source modules:

source module	source file	code location	written in
EXTH	HPIB	HPIB	Assembly
INIT_HPIB	H_DRV	HPIB	Pascal
HPIB_INITIALIZE	H_DRV	HPIB	Pascal

As stated earlier, there is no export text anywhere in the system for the driver modules. `HPIB_INITIALIZE` is the executable module for setting up the HP-IB driver. `INIT_HPIB` contains the procedure used to set up the driver. `EXTH` contains the low level code.

Some hardware notes about HP-IB. The HP-IB internal interface and the plug in 98624 interface are based on the Texas Instruments 9914 IEEE-488 chip. In general the interfaces are identical. The exceptions are that the internal HP-IB interface can only operate with DMA channel 0 and the internal interface does not have an ID register. Another deviation is that most cards fall on xxx0000 hex address boundries so you would expect the internal card to exist at 470000 because it is hardwired at internal select code 7, but it does not. The internal HP-IB resides at hex 478000.

The code in `EXTH` is based on the 9826 HPL system. There have been many modifications, primarily in the transfer mechanisms. Because of being taken from an HPL language system, there are some unusual pieces of code. The HPL system emulates the 9825 interface system. This means that the code attempts to make the HP-IB interfaces look like the 98034 interface cards on the 9825/35/45. This is where the 'eirbyte' came from -- the 9803x interface structure. The interrupts supported inside the code are 98034 style interrupts.

The 9914 works in a fairly straight forward way when it is used as a simple device or as a permanent system and active controller. When it is used as a general controller -- with selectable system control and passable active control -- things get more complex.

One of the aspects that makes the hpib drivers somewhat messy is the 'fakeisr' facility. Unless you are really interested in getting into the hpib drivers or you are writing your own hpib drivers understanding this is not necessary. The 'fakeisr' facility is when normal non-interrupt code is executing and this code notices that a hardware interrupt was missed. This normal code is then required to call the 'fakeisr' routine (`H_FAKEISR`) to get the driver to handle the missed interrupt (pretend an interrupt happened). This is necessary because when the drivers check the `INTOSTAT` register in the TI 9914 chip, it is cleared. When a condition occurs (SRQ, data in ready, or whatever) the enabled bit is set in the `INTOSTAT` register. If this register is read quickly enough, there is no interrupt generated and it is up to the code that read the register to handle the condition. The way that the drivers handle this is to see if any conditions occurred that it does not care about. If these conditions occurred, the `H_FAKEISR` routine is called (simulating a real interrupt). Eventually the fakeisr finishes and returns to the calling code. Any conditions that the fakeisr did not handle are placed in a copy location. The code then checks these conditions for the condition it was looking for. Note that there is a similar -- alternate problem where the 9914 can generate an ISR request and then have `INTOSTAT` read (removing the reason but not the interrupt). This causes a spurious ISR in the system -- no ISR will get execution but the system will try to poll the interfaces.

GPIO

The GPIO low level drivers consist of three source modules:

source module	source file	code location	written in
EXTG	GPIO	GPIO	Assembly
INIT_GPIO	G_DRV	GPIO	Pascal
GPIO_INITIALIZE	G_DRV	GPIO	Pascal

As stated earlier, there is no export text anywhere in the system for the driver modules. `GPIO_INITIALIZE` is the executable module for setting up the `HP-IB` driver. `INIT_GPIO` contains the procedure used to set up the driver. `EXTG` contains the low level code.

The GPIO card has a DMA priority switch. This switch does not have any direct effect on the DMA hardware. It is read by the driver firmware and is used to effect the DMA priority. The `dma_priority` field in the buffer control block and the switch are inclusively ORed.

The code in `EXTG` is based on the 9826 HPL system. There have been many modifications, primarily in the transfer mechanisms. Because of being taken from an HPL language system, there are some unusual pieces of code. The HPL system emulates the 9825 system. This means that the code attempts to make the GPIO interface look like the 98032 interface card on the 9825/35/45. Fortunately, the 98622 really does look like the 98032 interface.

DMA

The DMA drivers are, as a separate module, very simple. Mostly what the DMA drivers do is to look for the actual DMA hardware and set up some ISRs.

Some of the code for the support of DMA is in `IOCOMASM` -- the test, request and release channel routines. This code is relatively small.

It is necessary to have a special termination routine set up by the actual transfer drivers to catch the DMA interrupt. This is necessary because the transfer has two different terminating conditions -- the DMA count termination and whatever card terminations (like `HPIB eoi` termination).

Note that the DMA channels are actually two separate entities. Each channel has its own interrupt service routine.

The I/O library and drivers takes care of handling the DMA channels. If you are writing your own drivers and wish to use the DMA hardware there are some steps you must take to insure you do not conflict with the rest of the drivers. The DMA channels are resources that are allocated to various requestors by an algorithm. The requestor is viewed as being an interface card.

The algorithm is: If the requestor is the internal `HP-IB` card, then if channel 0 is available it is allocated to the `HP-IB` interface, otherwise no channel is allocated. (This is due to the fact that the internal `HP-IB` interface is not symmetrical with DMA channels.) If the requestor is any other interface, then channel 1 is allocated if available, otherwise channel 0 is allocated if available, otherwise no channel is allocated.

If you need to use DMA resources as part of your drivers, you must use the routines DMA_REQUEST and DMA_RELEASE in the module IOCOMASM (which is in the kernel). The form of use is:

```

.
.
.
VAR mychannel : INTEGER;
BEGIN
  mychannel := DMA_REQUEST(ISC_TABLE[ isc ].io_tmp_ptr^);

  { if mychannel is      -1 then I did not get a channel
                        0  then I got channel 0
                        1  then I got channel 1 }

  IF mychannel = -1
  THEN BEGIN
    { .....error or try again..... }
  END
  ELSE BEGIN
    { .....use the channel..... }
    DMA_RELEASE(ISC_TABLE [ isc ].io_tmp_ptr^);
  END;
.
.
.

```

Data Comm

The interface card contains a Z80, ROM, RAM, timers, and so on. It is a 'smart' card. It can support several different types of applications. Current configurations include asynchronous data communication, HP factory data link (DSN/DL), and shared resource manager uses. The shared resource manager use requires some hardware modifications. The drivers for data comm are actually a generic set of drivers for various configurations of the interface. The drivers, as they exist, support async, FDL, and SRM uses.

The data comm interface drivers are rather unusual. The assembly language driver code looks nothing at all like the 'atomic' operations. The code is based on the 98628 drivers in the BASIC language system. The code is designed specifically for the needs of the data comm card, not for the system that is using it. The data comm drivers consist of four modules:

source module	source file	code location	written in
EXTDC	DC	DATA_COMM	Assembly
INTDC	DC_DRV	DATA_COMM	Pascal
DC_INITIALIZE	DC_DRV	DATA_COMM	Pascal
INIT_DC	DC_DRV	DATA_COMM	Pascal

EXTDC is the low level assembly language drivers. INTDC is the Pascal that insulates the I/O structure and the low level structure. DC_INITIALIZE is the executable module that sets up the drivers. The EXTDC module consists of the following entry points:

ALVINIT	Card/driver initialization _ called once at power up. All further resets occur via a direct_control reset.
ALVINISR	Driver ISR.
ENTER_DATA	Read a block of data of specified length.
OUTPUT_DATA	Output a block of data.
OUTPUT_END	Output an 'end' control block _ like a CR/LF character pair or drop a modem line or send a special block on an FDL interface.
DIRECT_STATUS	Card status routine.
DIRECT_CONTROL	Card control routine _ do not wait in a queue _ do it immediately.
BFD_CONTROL	Card control routine _ queue up control command.
START_TFR_IN	Start an inbound transfer.
START_TFR_OUT	Start an outbound transfer.

A set of 'atomic' operations are implemented with these routines. These 'atomic' operations are:

IDC_INIT	driver initialization
IDC_ISR	driver isr
IDC_RDB	read a byte
IDC_WTB	write a byte
IDC_RDW	read a word
IDC_WTW	write a word
IDC_RDS	direct status
IDC_WTC	buffered and direct control
IDC_TFR	transfer

I/O Examples

Using Special Buffers

The I/O Library's transfer facility is oriented around character transfers. This is adequate for many needs, but by no means all the needs of a programmer. It is possible to trick the system into using another structure for the actual buffer data space. The steps involved are:

1. Create a buffer control variable.
2. Allocate an iobuffer with 0 bytes.
3. Change the buffer data space pointer to your structure.
4. Set the buffer size.
5. Reset the buffer.
6. Use the buffer.

The use of the buffer involves setting empty and fill pointers. As you take data from the buffer, increment the empty pointer. As you put data into the buffer, increment the fill pointer. If, in your application, you know how much data is coming in or going out you can just set the buffer empty or full before you do any I/O library transfers.

```
$SYSPROG ON$
PROGRAM specialbuffer(INPUT,OUTPUT);
IMPORT iodeclarations,general_4;
TYPE   short_integer = -32768..32767;
VAR    buffer : buf_info_type;
        stuff  : PACKED ARRAY[0..1023] OF short_integer;
        i,j    : INTEGER;
BEGIN
  iobuffer(buffer,0);                { set up for 0 bytes }
  WITH buffer DO BEGIN
    buf_ptr := ADDR(stuff);          { set up ptr to data }
    buf_size:= 2048;                 { size in bytes   }
  END; { of WITH DO BEGIN }

  FOR j:=0 TO 7 DO BEGIN

    FOR i:=0 TO 1023 DO stuff[i]:=i;  { put data into array }

    WITH buffer DO BEGIN
      buffer_reset(buffer);           { to get empty/fill set }
      buf_fill := ADDR(buf_fill,2048); { mark buffer full }
    END; { of WITH DO BEGIN }

    transfer(701,serial_fastest,from_memory,buffer,2048);
                                          { send data       }
  END; { of FOR DO BEGIN }
END. { of PROGRAM }
```

Remote Console Driver

This section is intended to show, by example, how to replace the system keyboard/crt drivers and install drivers for a remote console. Included is a functional example by which you can totally replace the existing console drivers with a remote console on an HP terminal connected via an RS-232 interface (either the 98626 or 98628 interface). If you want to do something other than this sample approach, you need to have familiarity with:

1. KBD modules in INITLIB.
2. Access methods.
3. I/O drivers and their structure.
4. CTABLE.
5. MISCINFO.

Before you do ANYTHING discussed in this section be sure you make back up copies of your BOOT disc (with INITLIB and TABLE) and of your CTABLE source.

There are two main approaches to putting in a remote console driver. The first is to merely add two new modules to the KBD part of INITLIB. This has the advantage of still allowing some 9826/36 interaction on the normal keyboard. The other approach is to replace part of the KBD modules with new drivers. This approach has the advantage of being less code in INITLIB but it does not allow ANY use of the normal keyboard.

There are a specific set of operations that need to happen to create a Pascal system with a remote console. These steps are:

1. BACK UP

Back up your BOOT disk and your CTABLE source.

2. CREATE NEW DRIVERS

Create a remote console (input and output) set of access modules (via the EDITOR and COMPILER). These modules correspond to the KEYS and CRT modules that contain the routines KBDIO and CRTIO.

3. INSTALL DRIVERS IN INITLIB

Install these modules in INITLIB on your boot disk (via the LIBRARIAN). The modules can either be added to the existing INITLIB modules or they can replace the current modules (i.e. KEYS and CRT).

4. MODIFY CTABLE

Change the TABLE file on your boot disk to make use of these new modules. This is done by editing CTABLE and compiling it and then copying the object file onto the TABLE file on the boot disk.

5. ADD MISCINFO TO BOOT

Change the system information about the console device with a MISCINFO file. This is done by compiling and running the MISCINFO program. This program will put a MISCINFO data file onto the boot disk.

Create New Drivers

The keyboard/CRT modules are actually a set of 5 modules. Each module is a separate program and exported module. The program part takes care of initializing the exported module. The modules are:

module	purpose	normally requires
KBD	fundamental support of the 'keyboard' 8041 uP including interrupt handling	
KEYS	support of the keyboard part of the keyboard	KBD
CRT	support of the CRT	KBD,KEYS
BAT	support of the battery part of the 'keyboard'	KBD
CLOCK	support of the timers part of the 'keyboard'	KBD

If CLOCK and BAT are intended to work normally (and actually function with clock and battery operations), then the KBD module needs to be fully functional. If battery and clock functions are not necessary (or are provided by some other means), then almost all of the modules can be replaced by dummy modules. The example at the end of this section shows the case where all five modules have been replaced.

There are three aspects of the KEYS and CRT modules that are a little strange and need some explanation. The first is the EOL_LYING_AROUND array in the KEYS module. The original code has an operation called READTOEOL. This operation is supposed to read all characters from the keyboard up to but NOT INCLUDING the EOL character (which is a carriage return). In the original code, there is a keyboard buffer that contains the characters. To read to EOL with the buffer you just look into the buffer until you find an EOL and back up one character. It is very difficult to push a character back into an interface. To accommodate this, the remote KEYS module will detect when a READTOEOL operation is in effect and an EOL is encountered and then set a flag. When the next input operation occurs, it checks to see if the EOL_LYING_AROUND flag is set. The EOL_LYING_AROUND flag is an array so that you can use these drivers for more than just the SYSTEM and CONSOLE volumes of the system.

The second strange aspect of the code is the NEWDRIVERS variable in KEYS and in CRT. This driver table contains a set of modified I/O drivers. The intent is to take the normal I/O drivers and remove the ability to reset the interface. This is necessary because many of the RS-232 line characteristics are set up via software but modified if a reset occurs. As a case in point, the 98628 interface needs to have control register 28 set to 0 to specify that there are no inbound

eol characters. If you did not do this, the interface would use the default of 2 characters for eol with those characters being <CR> and <LF>. Whenever a <CR> would come in from the terminal, the 98628 interface will NOT pass the <CR> on to the desktop computer because it is waiting to see if the next character is a <LF> and thereby completing the eol sequence. The interface must never be reset or the card will go back to its default 2 character eol sequence. The drivers must be modified because you can not depend on when a reset will occur - the IOINITIALIZE, IOUNINITIALIZE, and IORESET procedures and the STOP and CLR I/O keys will case this type of reset.

The third strange aspect of the drivers relates to CTABLE and INITUNITS. Before TABLE has had a chance to execute, messages are written to the CRT. The module INITUNITS initializes a minimum TABLE (CTABLE) to handle the definitions. It would be possible to change this module to specify the correct interface. In the example drivers a different approach was taken. The default TABLE (CTABLE) and INITUNITS specifies a select code of 0 for the CONSOLE and SYSTERM devices. The example drivers make use of this and the fact that external interface cards can only be on select codes of 8 and above. The code contains a line:

```
IF myisc <= 7 THEN myisc := 20;
```

This line will re-direct the I/O to select code 20. If you think about this for a bit, you will notice that you do not need to change CTABLE unless you are going to use more than one device as a remote volume. It might be desirable to change this code to search the interfaces for the first RS-232 interface that is in the desktop computer - it depends on your application.

Install Drivers in INITLIB

The modules, once they are compiled, need to be placed into INITLIB. The console modules should be in linked form to minimize the space they consume on the boot disk. For each of the modules that you are replacing (KBD, KEYS, CRT, BAT and/or CLOCK), go into the LIBRARIAN and link the compiled object file into a single module. For example for the KEYS module you would go through the following steps:

step	keystrokes	meaning
1.	CNEWKBD <cr> N <cr>	Go into the compiler and compile the source NEWKBD with no listing and put object code into NEWKBD.CODE
2.	LONEWKBD <cr> LINEWKBD <cr> ALKQ	Go into the librarian and specify an output file of NEWKBD.CODE link together all the modules of input file NEWKBD.CODE finishing linking, keep the output file and quit

Once you have all the modules you wish to replace in this linked form, you need to put them into INITLIB. To do this, it works best to create a temporary INITLIB (with a name of something like 'MYINIT.CODE') on a larger mass storage device. Go through and replace (or add) the

modules with the LIBRARIAN. The KBD, KEYS, etc. modules are some of the first modules in INITLIB. When you have replaced (or added) the appropriate modules, then keep the new temporary MYINIT and exit the LIBRARIAN. Go into the FILER and transfer the temporary MYINIT onto the BOOT disk with a file name of 'INITLIB.'

Modify Ctable

The CTABLE file needs to be modified to allow the remote console to work. The normal CTABLE (as shipped with the default system) will specify where the default CONSOLE and SYSTEMM volumes exist and what type of units they are.

The CTABLE changes to allow for the addition of the new remote console drivers look like the following:

```

procedure tea_crt(un:unitnum);
begin
  tea(un, 'MISC_UNBLOCKEDDAM',
      'REMC_CRTIO',           { change drv }
      21,                    { change isc }
      0,0,0,0,0, 'CONSOLE', #0,T,T,F,0);
end;

procedure tea_kbd(un:unitnum);
begin
  tea(un, 'MISC_UNBLOCKEDDAM',
      'REMK_KBDIO',         { change drv }
      21,                    { change isc }
      0,0,0,0,0, 'SYSTEMM', #0,F,T,F,0);
end;

```

The CTABLE changes to allow for the replacement of the remote console drivers look like the following:

```

procedure tea_crt(un:unitnum);
begin
  tea(un, 'MISC_UNBLOCKEDDAM', 'CRT_CRTIO',
      21,                    { change isc }
      0,0,0,0,0, 'CONSOLE', #0,T,T,F,0);
end;

procedure tea_kbd(un:unitnum);
begin
  tea(un, 'MISC_UNBLOCKEDDAM',
      'KEYS_KBDIO',         { change isc }
      21,
      0,0,0,0,0, 'SYSTEMM', #0,F,T,F,0);
end;

```

The CTABLE also has an alternate field that can be used for optional parameters (like baud rate or whatever). The alternate parameter is the parameter right before the volume name (e.g. 'SYSTEMM').

The approach taken with the remote console is such that these drivers can be used with other volumes in the system. Whether or not you wish to use the drivers as a remote console, it is possible to use the new KEYS and CRT modules as a general remote interface. Once the modules are placed in INITLIB, add the volumes to CTABLE source (and TABLE object file on the boot device).

Add MISCINFO to BOOT

The MISCINFO file needs to exist and specify an external CRT. Refer to the section on MISCINFO for more information.

Other Possibilities

It is also possible to use interfaces other than the serial interfaces shown in this example. Appropriate changes in KEYS and CRT will be necessary for the IOSTATUS and IOCONTROL usage. If you use an addressed interface (like HP-IB or HP-IL) it will also be necessary to preface the operations with a talk address or listen address sequence (assuming your interface is system/active controller).

In addition to using interfaces, it is possible to use no interface for the keyboard/crt device. This might be useful in a stand-alone application where no user interaction occurs. It is even possible to have the KEYS module contain sufficient information to send characters to the system (i.e. it sends a sequence of characters like '<cr><cr>FP#3<cr>QXmyprog<cr>' which would prefix the system to volume #3 and then execute the file 'myprog' on #3). In essence this could function similarly to Stream files.

Problems and Trouble Spots

There are some potential problems with dealing with a remote console. Some of these are:

Area	Problem
DEBUGGER	The debugger is hardwired to the internal CRT and keyboard of the 9826/36. You must leave the old KEYS and CRT module installed in the system if you intend to use the debugger and it must be used on the normal keyboard and CRT. Without re-writing the debugger, it is impossible to use from the remote console.
Stop key	The stop key can be supported in a limited way with the KEYS module. Currently, no support is included. It is possible to add stop key facilities in two ways. The first is to do an ESCAPE(-20) whenever a specific key is read from the interface. This approach depends on the keystrokes being read before the stop action occurs. The second approach is to use the SERIAL_5 interrupt facilities described elsewhere in this document to generate an interrupt when a BREAK occurs from the terminal. The ISR procedure that you install will then do an ESCAPE(-20) to cause the stop action.

Graphics It is not intended or possible with the 2.0 system to be able to do remote console (on screen) graphics. It will be possible in the 2.1 release to install new drivers into the graphics system to use a remote terminal.

HP9920 The keyboard interface must be installed, even if it is not being used. There is some part of the kernel of the Pascal system that is still depending on its existence.

There are some potential problems involved in trying to bring up the remote console example. Some of these are:

1. AUTO LF should be OFF

HP terminals respond to cursor sense differently when AUTO LF is enabled.

2. RS-232 CHARACTERISTICS

Make sure RS-232 line characteristics are the same. This includes:

baud rate
parity
stop bits
character or hardware handshakes (probably none)

3. ELECTRICAL CONNECTIONS

In most RS-232 hardware the lines are connected properly. However, just because the male and female RS-232 connectors can be connected physically does not mean they are electrically connected. A case in point is the HP 2382 terminal and the HP 98626/98628 option 001 RS-232 cable. The option 001 cable and terminal connected physically but pins 2 and 3 were turned around. It was necessary to wire up a special connector.

In general, the interface pins 1, 2, 3, and 7 are the fundamental lines.

4. TERMINAL TYPE

The examples are written with HP terminals in mind. The primary facility that is depended upon is the cursor sensing and cursor positioning facilities. If your terminal does not support the SAME mechanisms you will have to modify the programs appropriately.

Standard ASCII Keystroke Meanings

internal keyboard	ASCII	terminal keyboard
ENTER	CR	RETURN
up arrow	US	CTRL DEL
down arrow	LF	CTRL J
left arrow	BS	CTRL H
right arrow	FS	CTRL \
BACKSPACE	BS	BACKSPACE
space bar		space bar
EXECUTE		CTRL C
shift EXECUTE	ESC	ESC

```

$SYSPROG ON$
$heap_dispose off$
$iocheck off$
$range off$ $ovflcheck off$
$debug off$
$STACKCHECK OFF$

PROGRAM installkbd;

MODULE kbd;

IMPORT sysglobals,asm,bootdammodule,isr,misc;

EXPORT
  TYPE

  crtconsttype = PACKED ARRAY [0..11] of BYTE;

  CRTFREC = PACKED RECORD
    NOBREAK,STUPID,SLOWTERM,HASXYCRT,
    HASLCCRT{built in crt},HASCLOCK,
    canupscroll,candownscroll      :   BOOLEAN;
  END;

  B9 = PACKED ARRAY[0..8] OF BOOLEAN;
  B14= PACKED ARRAY[0..13] OF BOOLEAN;
  CRTCREC = PACKED RECORD          (* CRT CONTROL CHARS *)
    RLF,NDFS,ERASEEOL,
    ERASEEOS,HOME,
    ESCAPE           : CHAR;
    BACKSPACE       : CHAR;
    FILLCOUNT      : 0..255;
    CLEARSCREEN,
    CLEARLINE       : CHAR;
    PREFIXED        : B9
  END;

  CRTIREC = PACKED RECORD          (* CRT INFO & INPUT CHARS *)
    WIDTH,HEIGHT    : shortint;
    crtmemaddr,crtcontroladdr,
    keybufferaddr,progstateinfoaddr: INTEGER;
    keybuffersize: shortint;
    crtcon          : crtconsttype;
    RIGHT,LEFT,DOWN,UP: CHAR;
    BADCH,CHARDEL,STOP,
    BREAK,FLUSH,EOF : CHAR;
    ALTMODE,LINEDEL : CHAR;
    BACKSPACE,
    ETX,PREFIX      : CHAR;
    PREFIXED        : B14 ;
    CURSORMASK     : INTEGER;
    SPARE           : INTEGER;
  END;

  ENVIRON = RECORD
    MISCINFO: CRTFREC;

```

```

        CRTTYPE:  INTEGER;
        CRTCTRL:  CRTCREC;
        CRTINFO:  CRTIREC;
    END;

stat8041 = PACKED RECORD
    case INTEGER of
        0: (pad1:    0..63;
           busy:    BOOLEAN;
           readready:BOOLEAN);
        1: (statchar: CHAR);
    END;
crtword=  RECORD case INTEGER of
        1:(highlightbyte,character:CHAR;);
        2:(wholeword: shortint);
    END;
kbdhooktype = PROCEDURE(VAR statbyte,databyte: BYTE;
                        VAR dokey: BOOLEAN);

timerhooktype = PROCEDURE(statbyte,databyte: BYTE;
                          VAR dotimer: BOOLEAN);

keybuffertype= ARRAY[0..maxint] of crtword;

```

VAR

```

SYSCOM: ^ENVIRON;
changehardware: BOOLEAN;
progstateinfo:^keybuffertype;

ALPHASTATE['ALPHAFLAG']:BOOLEAN;
GRAPHICSTATE['GRAPHICSFLAG']:BOOLEAN;

kbdhook: kbdhooktype;
timerhook: timerhooktype;
dumpalphahook: PROCEDURE;
dumpgraphicshook: PROCEDURE;
togglealphahook: PROCEDURE;
togglegraphicshook: PROCEDURE;

kbeepfreq, kbeepdur : BYTE;

PROCEDURE beep;
PROCEDURE beeper(frequency,duration : BYTE);

PROCEDURE kbdinit;
PROCEDURE lockedaction(a: action);

PROCEDURE kbdcommand(cmd      : BYTE;
                    numdata   : INTEGER;
                    b1, b2, b3 : BYTE);
FUNCTION read8041byte : BYTE;

```

IMPLEMENT

CONST

```
B9826INFO=CRTIREC[
    WIDTH           : 80,HEIGHT:24,
    crtmemaddr      : 5316608,
    crtcontroladdr  : 5341185,
    keybufferaddr   : 5320448,
    progstateinfoaddr : 5320592,
    keybuffersize   : 72,
    crtcon          : crtconsttype [114,80,76,7,
                                   26,10,25,25,
                                   0,14,76,13],

    RIGHT{FS}      : CHR(28),
    LEFT{BS}       : CHR(8),
    DOWN{LF}       : CHR(10),
    UP{US}         : CHR(31),
    BADCH{?}       : CHR(63),
    CHARDEL{BS}    : CHR(8),
    STOP{DC3}      : CHR(19),
    BREAK{DLE}     : CHR(16),
    FLUSH{ACK}     : CHR(6),
    EOF{ETX}       : CHR(3),
    ALTMODE{ESC}   : CHR(27),
    LINEDEL{DEL}   : CHR(127),
    BACKSPACE{BS}  : CHR(8),
    ETX            : CHR(3),
    PREFIX         : CHR(0),
    PREFIXED       : B14[14 OF FALSE],
    CURSORMASK     : 0,
    SPARE          : 0];
```

CONST

```
ENVIRONC=ENVIRON[MISCINFO:CRTFREC[
    NOBREAK : FALSE,
    STUPID  : FALSE,
    SLOWTERM : FALSE,
    HASXYCRT : TRUE,
    HASLOCRT : TRUE, {?}
    HASCLOCK : TRUE,
    canupscroll : TRUE,
    candownscroll : TRUE],

CRTTYPE:0,
CRTCTRL : CRTCREC[
    RLF      : CHR(31),
    NDFS     : CHR(28),
    ERASEEOL : CHR(9),
    ERASEEOS : CHR(11),
    HOME     : CHR(1),
    ESCAPE   : CHR(0),
    BACKSPACE: CHR(8),
    FILLCOUNT: 10,
    CLEARSCREEN: CHR(0),
    CLEARLINE: CHR(0),
    PREFIXED : B9[9 OF FALSE]],

CRTINFO : CRTIREC [
    WIDTH           : 50,
```



```

HEIGHT          : 24,
crtmemaddr      : 5316608,
crtcontroladdr  : 5308417,
keybufferaddr   : 5319008,
progstateinfoaddr: 5319092,
keybuffersize   : 42,
crtcon: crtconsttype [64,50,49,10,25,9,
                      25,25,0,11,74,11],
RIGHT{FS}       : CHR(28),
LEFT{BS}        : CHR(8),
DOWN{LF}        : CHR(10),
UP{US}          : CHR(31),
BADCH{?}        : CHR(63),
CHARDEL{BS}     : CHR(8),
STOP{DC3}       : CHR(19),
BREAK{DLE}      : CHR(16),
FLUSH{ACK}      : CHR(6),
EOF{ETX}        : CHR(3),
ALTMODE{ESC}    : CHR(27),
LINEDEL{DEL}    : CHR(127),
BACKSPACE{BS}   : CHR(8),
ETX             : CHR(3),
PREFIX          : CHR(0),
PREFIXED        : B14[14 OF FALSE],
CURSORMASK      : 0,
SPARE           : 0]];

```

```

PROCEDURE lockedaction(a: action);
label 1;
VAR i: INTEGER;
BEGIN
  IF locklevel = 0 THEN call(a)
  ELSE
    BEGIN
      i := actionspending;
      WHILE i>0 DO IF deferredaction[i]=a THEN goto 1 ELSE i := i - 1;
      IF actionspending = 10 THEN beep
      ELSE BEGIN
        actionspending := actionspending + 1;
        deferredaction[actionspending] := a;
      END;
    END;
  1;
END;

FUNCTION read8041byte:BYTE;
BEGIN
  read8041byte:=0;
END;

PROCEDURE wait4kbdready;
BEGIN
END;

```

```

PROCEDURE kbdcommand(cmd      : BYTE;
                    numdata   : INTEGER;
                    b1, b2, b3 : BYTE);
BEGIN
END;

PROCEDURE beep;
BEGIN
END;

PROCEDURE beeper(frequency,duration : BYTE);
BEGIN
END;

PROCEDURE dummykbdhook(VAR stat, data: BYTE;
                      VAR doit: BOOLEAN);
BEGIN
END;

PROCEDURE dummytimerhook(stat, data: BYTE;
                          VAR doit: BOOLEAN);
BEGIN
END;

PROCEDURE INITSYSCOM;
VAR f: file of ENVIRON;
    dcrinfo['dcrinfo']: anyptr;
BEGIN
NEW(SYSCOM); SYSCOM^ := ENVIRONC;
WITH syscom^ DO
  BEGIN
    IF not sysflag.alpha50 THEN crtinfo := B9826info;
    RESET(F, NODESTR+'MISCINFO','shared');
    IF IORESULT = ORD(INOERROR) THEN READ(F, SYSCOM^);
    changehardware := IORESULT = ORD(INOERROR);
    dcrinfo := ADDR(crtinfo);
  END;
END; {INITSYSCOM}

PROCEDURE kbdinit;
BEGIN
  kbdhook := dummykbdhook;
  timerhook := dummytimerhook;
  initsyscom;
END; {kbdinit}

END; { of module }

IMPORT kbd;

```

```
BEGIN  
  kbdinit;  
END.
```

```

$SYSPROG ON$
$heap_dispose off$
$iocheck off$
$range off$ $ovflcheck off$
$debug off$
$STACKCHECK OFF$

PROGRAM installkeys;

MODULE keys;
IMPORT sysglobals,asm,misc,kbd, iodeclarations,general_0,iocomasm;

EXPORT
CONST
    yencode = 92; { Yen symbol overlays USASCII
                   backslash (\) in Kana machines }

TYPE
    langtype = (gringo,french,german,swedish{,finnish},
               spanish,katakana);

VAR
    kbdlangjumper: RECORD CASE BYTE of
        0: (b:PACKED RECORD
            dummy,jnum:BYTE
            END);
        1: (jlang:langtype); {16 bit}
    END;
    kbdwaithook      : PROCEDURE;
    kbdreleasehook   : PROCEDURE;

    keybuffer        : ^keybuffertype;
    keybufsize        : shortint;
    keybuflength     : shortint;
    capslock          : BOOLEAN;
    kanaflag          : BOOLEAN;

PROCEDURE kbdio (    fp           : fibp;
                   request       : amrequesttype;
                   ANYVAR buffer : window;
                   length        : INTEGER ;
                   position      : INTEGER);

PROCEDURE initkeys;

IMPLEMENT

VAR eol_lying_around : PACKED ARRAY[type_isc] OF BOOLEAN;
    myisc             : shortint;

    newdrivers        : drv_table_type;

```

```
{ note that you should not use the 'console'  
  select code for anything else }
```

```
PROCEDURE new_reset(mytemp : ANYPTR);  
BEGIN  
  { do nothing so that the configuration stays the same }  
END;
```

```
PROCEDURE myinit;  
BEGIN  
  IF isc_table[myisc].card_id = hp98628_async  
    THEN iocontrol(myisc,28,0);           { no EOL characters }  
    iocontrol(myisc,12,1);                 { connect the card }  
  
  newdrivers := isc_table[myisc].io_drv_ptr^; { copy card drvrs }  
  newdrivers.iod_init := new_reset;         { put in new reset }  
  isc_table[myisc].io_drv_ptr := ADDR(newdrivers); { install drvrs }  
END;
```

```
FUNCTION inchar : CHAR;  
VAR    x      : CHAR;  
BEGIN  
  IF eol_lying_around[myisc]  
    THEN BEGIN  
      inchar := eol;  
      eol_lying_around[myisc] := FALSE;  
    END  
    ELSE BEGIN  
      WITH isc_table[myisc] DO  
        CALL (io_drv_ptr^.iod_rdb ,  
              io_tmp_ptr ,  
              x);  
      inchar:=x;  
    END;  
END;
```

```
FUNCTION kbdbusy : BOOLEAN;  
VAR    x      : INTEGER;  
BEGIN  
  IF isc_table[myisc].card_id = hp98628_async  
    THEN BEGIN  
      { check inbound queue for data }  
      x:=iostatus(myisc,5);  
      IF (x=1) OR (x=3) OR eol_lying_around[myisc]  
        THEN kbdbusy:=FALSE  
        ELSE kbdbusy:=TRUE;  
    END;  
  IF isc_table[myisc].card_id = hp98626
```

```

THEN BEGIN
  x:=iostatus(myisc,10);
  { check character buffer for data }
  IF bit_set(x,0) OR eol_lying_around[myisc]
    THEN kbdbusy:=FALSE
    ELSE kbdbusy:=TRUE;
END;
END;

```

```

PROCEDURE kbdio (      fp          : fibp;
                     request      : amrequesttype;
                     ANYVAR buffer : window;
                     length       : INTEGER ;
                     position     : INTEGER);

```

```

VAR  buf          : charptr;

```

```

BEGIN

```

```

  myisc := unitable^[fp^.funit].sc;

```

```

  IF myisc <= 7 THEN myisc := 20;

```

```

  ioreult := ORD(inoerror);

```

```

  buf := ADDR(buffer);

```

```

  CASE request OF

```

```

    flush:      BEGIN
                  myinit;
                END;

```

```

    unitstatus: BEGIN
                  fp^.fbusy := kbdbusy ;
                END;

```

```

    clearunit:  BEGIN
                  myinit;
                END;

```

```

    readtoeol,

```

```

    readbytes,

```

```

    startread:

```

```

      BEGIN

```

```

        IF request = readtoeol

```

```

          THEN BEGIN

```

```

            { the buffer is a string, so set it to empty }

```

```

            buf := ADDR(buf^, 1);

```

```

            buffer[0] := chr(0);

```

```

          END;

```

```

        WHILE length>0 DO BEGIN

```

```

          buf^ := inchar;

```

```

          IF buf^ = chr(etx)

```

```

            THEN length := 0

```

```

            ELSE length := length-1;

```

```

          IF (buf^=eol) and (request=readtoeol)

```

```

            THEN BEGIN

```

```

              eol_lying_around[myisc] := TRUE;

```

```

              length := 0

```

```

        END
        ELSE BEGIN
            fp^.feoln := FALSE;
            buf := ADDR(buf^, 1);
            IF request = readtoeol
                THEN buffer[0] := CHR(ORD(buffer[0])+1);
            END;
        END; { of WHILE DO }
        IF request = startread THEN CALL(fp^.feot, fp);
    END;

    OTHERWISE BEGIN
        ioreresult := ORD(ibadrequest);
    END;

END; { of CASE }
END; { of PROCEDURE }

```

```

PROCEDURE dummyproc;
BEGIN
    { nothing }
END;

```

```

PROCEDURE initkeys;
VAR localisc : shortint;
BEGIN
    FOR localisc := 0 TO 31 DO eol_lying_around[localisc] := FALSE;
    WITH syscom^.crtinfo DO BEGIN
        keybuffer:=NIL;
        keybufsize:=1;
        kanaflag:=FALSE;
        capslock:=TRUE;
    END;
END;

END;{ of module }

```

```

IMPORT keys;

BEGIN
    initkeys;
END.

```

```

$SYSPROG ON$
$heap_dispose off$
$iocheck off$
$range off$ $ovflcheck off$
$debug off$
$STACKCHECK OFF$

PROGRAM installcrt;

MODULE crt;
IMPORT sysglobals,asm,misc,kbd,keys, iodeclarations,general_0 ;
EXPORT
    TYPE scrtype = PACKED ARRAY[0..maxint] OF crtword;
        scrptr = ^scrtype;

    VAR screenwidth      : shortint;
        screenheight    : shortint;
        maxx,maxy      : shortint;
        screensize      : shortint;
        xpos,ypos       : shortint;
        screen          : scrptr;
        defaulthighlight : shortint;

    PROCEDURE crtinit;
    PROCEDURE crtio (    fp          : fibp;
                        request     : amrequesttype;
                        ANYVAR buffer : window;
                        length      : INTEGER;
                        position     : INTEGER);

    PROCEDURE updatecursor;

    PROCEDURE setrunlight(x:CHAR);

IMPLEMENT

CONST dc1          = 17 ;
VAR   myisc        : shortint;
      newdrivers   : drv_table_type;

{ note that you should not use the 'console'
  select code for anything else }

PROCEDURE new_reset(mytemp : ANYPTR);
BEGIN
    { do nothing so that the configuration stays the same }
END;

```



```

PROCEDURE myinit;
BEGIN
  IF isc_table[myisc].card_id = hp98628_async
    THEN iocontrol(myisc,28,0);           { no EOL characters }
  iocontrol(myisc,12,1);                 { connect the card }

  newdrivers := isc_table[myisc].io_drv_ptr^; { copy card drvrs }
  newdrivers.iod_init := new_reset;         { put in new reset }
  isc_table[myisc].io_drv_ptr := ADDR(newdrivers); { install drvrs }
END;

```

```

FUNCTION inchar : CHAR;
VAR x          : CHAR;
BEGIN
  WITH isc_table[myisc] DO
    CALL (io_drv_ptr^.iod_rdb ,
          io_tmp_ptr ,
          x);
  inchar:=x;
END;

```

```

PROCEDURE out(x:CHAR);
BEGIN
  WITH isc_table[myisc] DO
    CALL (io_drv_ptr^.iod_wtb ,
          io_tmp_ptr ,
          x);
END;

```

```

PROCEDURE output(s :io_STRING);
VAR i:INTEGER;
BEGIN
  FOR i:=1 to STRLEN(s) DO out(s[i]);
END;

```

```

PROCEDURE localbeep;
BEGIN
  out(CHR(7));          { send beep to card }
END;

```

```

PROCEDURE setrunlight(x:CHAR);
BEGIN
  { DO nothing at all but have an exported PROCEDURE }
END;

```

```

PROCEDURE updatecursor;
BEGIN
  { DO nothing at all but have an exported PROCEDURE }
END;

PROCEDURE getxy(VAR x,y: INTEGER);
VAR dummy : CHAR;
BEGIN
  x:=0; y:=0;
  { go thru sequence to get actual position }
  out(CHR(esc));          out('');          { send cursor sense abse  }
  out(CHR(dcl));          { tell terminal I am ready }
  dummy := inchar;        { get esc  }
  dummy := inchar;        { get &  }
  dummy := inchar;        { get '  }
  x := ORD(inchar)-48;    { get column digit 1 }
  x := ORD(inchar)-48+x*10; { get column digit 2 }
  x := ORD(inchar)-48+x*10; { get column digit 3 }
  dummy := inchar;        { get c  }
  y := ORD(inchar)-48;    { get row digit 1 }
  y := ORD(inchar)-48+y*10; { get row digit 2 }
  y := ORD(inchar)-48+y*10; { get row digit 3 }
  dummy := inchar;        { get Y  }
  dummy := inchar;        { get cr  }

  xpos := x;      ypos := y;
END;

PROCEDURE setxy(x, y: shortint);
VAR s : string[9];
    p : INTEGER;
BEGIN
  IF x>=screenwidth THEN xpos:=maxx
    ELSE IF x<0 THEN xpos:=0
      ELSE xpos := x;
  IF y>=screenheight THEN ypos:=maxy
    ELSE IF y<0 THEN ypos:=0
      ELSE ypos := y;

  { send xpos/ypos via escape esc & a xx y yy C }
  SETSTRLEN(s,9);
  STRWRITE (s,1,p,CHR(esc),'&a',ypos:2,'y',xpos:2,'C');
  output (s);
END;

PROCEDURE gotoxy(x,y: INTEGER);
BEGIN
  setxy(x,y);
  updatecursor;
END;

```

```

PROCEDURE crtio (      fp          : fibp;
                    request      : amrequesttype;
                    ANYVAR buffer : window;
                    length       : INTEGER;
                    position     : INTEGER);

VAR c      : CHAR;
    s      : STRING[1];
    buf    : charptr;
    d,e    : INTEGER;
BEGIN
  myisc := unitable^[fp^.funit].sc;
  IF myisc <= 7 THEN myisc := 20;
  ioresult := ORD(inoerror);
  buf := ADDR(buffer);
  CASE request OF

    setcursor:  BEGIN
                  gotoxy(fp^.fxpos, fp^.fypos);
                END;

    getcursor:  BEGIN
                  getxy (fp^.fxpos, fp^.fypos);
                END;

    flush:      BEGIN
                  myinit;
                END;

    unitstatus: BEGIN
                  kbdio(fp, unitstatus,buffer,length,position);
                END;

    clearunit:  BEGIN
                  myinit;
                END;

    readtoeol: BEGIN
                  buf := ADDR(buf^, 1);
                  buffer[0] := CHR(0);
                  WHILE length>0 DO BEGIN
                    kbdio(fp, readtoeol, s, 1, 0);
                    IF STRLEN(s)=0
                      THEN BEGIN
                           length := 0
                        END
                      ELSE BEGIN
                           length := length - 1;
                           crtio(fp, writebytes, s[1], 1, 0);
                           buf := ADDR(buf^, 1);
                           buffer[0] := CHR(ORD(buffer[0])+1);
                        END; { of IF }
                  END; { of WHILE DO BEGIN }
                END; { of BEGIN }

    startread,
    readbytes:  BEGIN
  
```

```

WHILE length>0 DO
  BEGIN
    kbdio(fp, readbytes, buf^, 1, 0);
    IF buf^ = CHR(etx) THEN length := 0
      ELSE length := length - 1;

    IF buf^ = eol
      THEN crtio(fp, writeeol, buf^, 1, 0)
      ELSE crtio(fp, writebytes, buf^, 1, 0);
    buf := ADDR(buf^, 1);
    END;
  IF request = startread THEN call(fp^.feot, fp);
  END;

writeeol: BEGIN
  IF ypos=maxy
    THEN BEGIN
      out(CHR(esc));
      out('S');           { scroll up 1 line }
    END;
  gotoxy(0, ypos+1);
  END;

startwrite,
writebytes: BEGIN
  WHILE length>0 DO BEGIN
    c:=buf^; buf:=ADDR(buf^,1); length:=length-1;
    CASE c OF

      homechar: BEGIN
        setxy(0,0);
        END;

      leftchar: BEGIN
        out(CHR(bs));
        END;

      rightchar: BEGIN
        getxy(d,e);
        IF (xpos = maxx) and (ypos<maxy)
          THEN setxy(0, ypos+1)
          ELSE setxy(xpos+1, ypos);
        END;

      upchar: BEGIN
        IF (ypos<=1)
          THEN BEGIN
            out(CHR(esc));
            out('L');     { insert line }
          END;
        IF (ypos>0)
          THEN BEGIN
            { out(CHR(esc));
              out('A'); }
            setxy(xpos,ypos-1);
          END;
        END;
    END;
  END;

```

```

downchar: BEGIN
    IF (ypos=maxy)
    THEN BEGIN
        out(CHR(esc));
        out('S'); { scroll up 1 line }
    END
    ELSE BEGIN
        { out(CHR(esc));
        out('B'); }
        setxy(xpos,ypos+1);
    END;
END;

bellchar: BEGIN
    localbeep;
END;

cteos: BEGIN
    out(CHR(esc));
    out('J');
END;

cteol: BEGIN
    out(CHR(esc));
    out('K');
END;

clearscr:BEGIN
    setxy(0,0);
    out(CHR(esc));
    out('J');
END;

eol: BEGIN
    out(CHR(cr));
    out(CHR(lf));
END;

CHR(ety): BEGIN
    length:=0;
END;

OTHERWISE BEGIN
    out(c);
    IF xpos = maxx
    THEN BEGIN
        IF ypos = maxy
        THEN BEGIN
            out(CHR(esc));
            out('S'); { scroll up 1 line }
        END;
        setxy(0,ypos+1);
    END
    ELSE BEGIN
        { setxy(xpos+1,ypos); }
        xpos := xpos + 1;
    END;
END;

```

```

                                END; { of IF }
                                END;

                                END; { of CASE c OF }
                                updatecursor;
                                END; { of WHILE DO BEGIN }
                                IF request = startwrite THEN call(fp^.feot, fp);
                                END; { of startwrite, writebytes case }

OTHERWISE BEGIN
    ioreresult := ORD(ibadrequest);
    END;

END; { of CASE request OF }
END; { of PROCEDURE crtio }

```

```

PROCEDURE dummyproc;
BEGIN
    { nothing }
END;

```

```

PROCEDURE crtinit;
BEGIN
    WITH syscom^.crtinfo DO BEGIN
        screen      :=NIL;
        screenwidth:=width;
        screenheight:=height;
        screensize  :=width*height;
        maxx       :=width-1;
        maxy        :=height-1;
        xpos        :=0;
        ypos        :=0;
        defaulthighlight := 0;
        dumpalphahook  := dummyproc;
        dumpgraphicshook := dummyproc;
        togglealphahook := dummyproc;
        togglegraphicshook := dummyproc;
        ALPHASTATE := TRUE;
    END; { of WITH DO BEGIN }
END; { of PROCEDURE crtinit }

```

```

END; { of MODULE crt }

```

```

IMPORT crt;

BEGIN
    crtinit;
END.

```

```

$SYSPROG ON$
$heap_dispose off$
$iocheck off$
$range off$ $ovflcheck off$
$debug off$
$STACKCHECK OFF$

PROGRAM installbat;

MODULE bat;
IMPORT sysglobals, kbd;
EXPORT
VAR batterypresent[-563]: BOOLEAN;

    PROCEDURE batcommand(cmd          : BYTE;
                          numdata     : INTEGER;
                          b1, b2, b3, b4, b5 : BYTE);
    FUNCTION batbytereceived:BYTE;
    PROCEDURE batinit;

IMPLEMENT

PROCEDURE batcommand(cmd          : BYTE;
                      numdata     : INTEGER;
                      b1, b2, b3, b4, b5 : BYTE);

BEGIN
END;

FUNCTION batbytereceived : BYTE;
BEGIN
    batbytereceived := 0;
END;

PROCEDURE batinit;
BEGIN
END;

END; { of MODULE }

IMPORT bat;

BEGIN
    batinit;
END.
^^F
$SYSPROG ON$
$heap_dispose off$
$iocheck off$
$range off$ $ovflcheck off$
$debug off$
$STACKCHECK OFF$

PROGRAM installclock;

```

```

MODULE clock;
IMPORT sysglobals, asm, kbd, bat;
EXPORT
  TYPE
    RTCTIME = PACKED RECORD
      PACKEDTIME,PACKEDDATE:INTEGER;
    END;

  FUNCTION sysclock: INTEGER; {centiseconds from midnight}
  PROCEDURE sysdate (VAR thedate: daterec);
  PROCEDURE systime (VAR thetime: timerec);
  PROCEDURE setsysdate (thedate: daterec);
  PROCEDURE setsystime (thetime: timerec);
  PROCEDURE initclock;

```

implement

```

PROCEDURE SYSDATE(VAR THEDATE: DATEREC);
BEGIN
  WITH THEDATE DO
    BEGIN
      YEAR:=00;
      MONTH:=01;
      DAY:=01;
    END;
END;

```

```

FUNCTION sysclock: INTEGER;
BEGIN
  sysclock := 0;
END;

```

```

PROCEDURE SYSTEM(VAR THETIME: TIMEREC);
BEGIN
  WITH THETIME DO
    BEGIN
      HOUR      := 00;
      MINUTE    := 00;
      CENTISECOND := 0000;
    END;
END;

```

```

PROCEDURE setsysdate(thedate: daterec);
BEGIN
END;

```

```

PROCEDURE setsystime(thetime: timerec);
BEGIN
END;

```

```

PROCEDURE inittime;
BEGIN
END;

```



```
PROCEDURE initclock;  
BEGIN  
END;  
  
END;  
  
IMPORT clock;  
  
BEGIN  
    initclock;  
END.
```

Removal of Drivers

The structure of the code is such that only the kernel (IODECLARATIONS in INITLIB) must be in INITLIB. The rest of the INITLIB drivers can be removed. The high level routines that exist in LIBRARY are there in relatively small modules so you only get what you need.

In general, the removal of drivers is necessary to create a minimum system disk for stand alone applications. As stated earlier the IODECLARATIONS module is about 3K bytes and all the I/O device drivers are slightly larger than 12K bytes. The drivers in INITLIB and their uses are:

driver	use
HPIB	Built in HP-IB interface and 98624 interfaces. Used by disk and printer drivers that go through these interfaces.
GPIO	For 98622 interfaces. Only system use is if the system table specifies a printer on a 98622 interface, then these drivers are used. The 9885 floppy disk drives have their own drivers and do not go through the GPIO module.
Data Comm	For 98628 and 98629 interfaces. The system will use these drivers if the system table specifies a printer (or other unblock volume) on a 98628 card. The system uses these drivers for the shared resource manager access.
DMA	If there is no DMA card in the system, this driver can be removed. The 9885 disk drivers require the DMA hardware and this driver to be present.
RS-232	For 98626 interfaces. Only system use is when the table specifies the interface as an unblocked volume.

Addition of a Driver

The general structure of a new driver follows the form shown in the existing drivers -- the low level driver code and some initialization code. The initialization code consists of the following pieces:

1. Set up the new driver table.
2. Search the `ISC__TABLE` for interfaces of my type.
3. If the code has any ISR support or needs, perform a permanent ISR linkage to the driver ISR.
4. Initialize the drivers and interface.

The driver code and the initialization need to be linked (via the librarian) and placed onto `INITLIB`. If this code does not allocate any heap space, it is possible to merely load it as a permanent library (via the 'P' command in the main command interpreter). This 'P' facility makes the debugging of the new drivers a lot easier.

A Specific Example

The following example is a simple 'dummy' driver. It shows the main aspects of a new driver from a structural point of view. It does not show the interrupt linkages. The code is designed so that it puts itself in at all select codes that have no __id.

```
$SYSPROG ON$
(*****)
(*)
(*)
(*)      IOLIB          example drivers
(*)
(*)
(*****)
(*)
(*)
(*)      library       -  IOLIB
(*)      name          -  DUMMY
(*)      module(s)    -  extd
(*)                  -  init_dummy
(*)                  -  dummy_initialize
(*)
(*)      date          -  July 21 , 1982
(*)      update        -  July 21 , 1982
(*)
(*****)
```

```
PROGRAM dummy_initialize (INPUT , OUTPUT);
    { This module has a program segment so that there is
      an executable entry point into the module.
      At INITLIB time this program is executed. }
```

```
MODULE extd;
IMPORT sysglobals , iodeclarations ;
EXPORT
    PROCEDURE ed_init (temp : ANYPTR);
    PROCEDURE ed_rdb (temp : ANYPTR ; VAR x : CHAR);
    PROCEDURE ed_wtb (temp : ANYPTR ; val : CHAR);
    PROCEDURE ed_send (temp : ANYPTR ; val : CHAR);
IMPLEMENT

    PROCEDURE ed_init (temp : ANYPTR);
    BEGIN
        WRITELN('INITIALIZATION on ',io_find_isc(temp):4);
    END;

    PROCEDURE ed_rdb (temp : ANYPTR ; VAR x : CHAR);
    BEGIN
        WRITELN('READ CHARACTER on ',io_find_isc(temp):4);
        READ(x);
    END;

    PROCEDURE ed_wtb (temp : ANYPTR ; val : CHAR);
    BEGIN
        WRITELN('WRITE CHARACTER on ',io_find_isc(temp):4);
```

```

    WRITE(val);
END;

PROCEDURE ed_send (temp : ANYPTR ; val : CHAR);
BEGIN
    WRITELN('SEND COMMAND on ',io_find_isc(temp):4,
            ' of command ',ORD(val):3);
END;
END; { of extd }

MODULE init_dummy ; { This module initializes the HPIB drivers. }
IMPORT iodeclarations ;
EXPORT
    CONST dummy_id = -100;
          dummy_type = 100;
    VAR my_dummy_drivers : drv_table_type;
    PROCEDURE io_init_dummy;
IMPLEMENT
    IMPORT sysglobals , isr , general_0 , extd ;

    PROCEDURE io_init_dummy;
    VAR io_isc : type_isc;
        dummy : INTEGER;
        io_lvl : io_byte;
    BEGIN
        io_revid := io_revid + ' DUMMY1.0'; { io_revid indicates
                                             what version of the
                                             drivers are in the
                                             system. }

        { set up the driver tables }
        WITH my_dummy_drivers DO BEGIN
            my_dummy_drivers := dummy_drivers; { sets up the table
                                                with all dummy
                                                entries }

            iod_init := ed_init;
            iod_rdb := ed_rdb;
            iod_wtb := ed_wtb;
            iod_send := ed_send;
        END; { of WITH }

        { set up drivers for the interfaces }
        FOR io_isc:=iomisc TO iomaxisc DO
            WITH isc_table[io_isc] DO BEGIN
                IF (card_id = no_id)
                THEN BEGIN
                    card_id := dummy_id; { put in my id }
                    card_type := dummy_type; { put in my type }
                    io_drv_ptr:=ADDR(my_dummy_drivers);
                    { link in an ISR here if it is necessary }
                END; { of IF card_id }
            END; { of FOR io_isc WITH isc_table[io_isc] BEGIN }

        { call the actual driver initialization }
        { this is separate from the set up code in case
          there are 2 or more cards connected - and generate

```

```
    an isr between each other }
FOR io_isc:=iominisc TO iomaxisc DO
  WITH isc_table[io_isc] DO
    IF (card_id = dummy_id)
      THEN BEGIN
        CALL(io_drv_ptr^.iod_init , io_tmp_ptr);
      END; { of WITH IF }
    END; { of io_init_dummy }
  END; { of MODULE init_dummy }
```

```
IMPORT    init_dummy ;
BEGIN
  io_init_dummy;
END. { of dummy_initialize }
```

Modification of a Driver

It is possible for a user to extend the drivers. You must create a driver table and fill it with appropriate procedures. Then you must modify the `isc__table` to point to the new driver table. An example might be to extend the the `kbd/crt` drivers to show the character values that are being sent to select code 1.

```
$SYSPROG ON$
PROGRAM modifydrivers(INPUT,OUTPUT);
IMPORT iodeclarations,general_1,general_2;
VAR    newkbd : drv_table_type;
        oldkbd : ^drv_table_type;
        i      : INTEGER;

    { new driver procedure }
    PROCEDURE MYPROC(mytemp : ANYPTR ;
                     mychar : CHAR);
BEGIN
    WRITELN('write byte of character value ',ORD(mychar):3,
            ' is <',mychar,'>');
END;

BEGIN
    { set up new drivers }
    newkbd := isc_table[1].io_drv_ptr^;      { to copy some drivers }
    oldkbd := ADDR(isc_table[1].io_drv_ptr^);{ to keep the old ones }
    newkbd.iod_wtb := MYPROC;                { add new procedures }
    isc_table[1].io_drv_ptr := ADDR(newkbd); { set up isc table [1] }

    { use new drivers }
    writenumberln(1,12.345);

    { remove new drivers }
    isc_table[1].io_drv_ptr := ADDR(oldkbd^);
                                                    { restore isc table [1] }
END.
```

End-of-Transfer Procedures

The transfer facility in the drivers supports an end-of-transfer (EOT) procedure. When a transfer completes, the specified procedure is called. This is very similar to the ON EOT capability in BASIC. One major difference is that the procedure is called from inside an ISR (since that is where the transfer was detected as finished). There is no end-of-line in Pascal, so you have to be very careful of the operations inside an EOT procedure.

The current library has no high level support for EOT. Rather than force the use of a rather unpleasant mechanism, the following module will provide a nice, high-level set of routines.

Note that you can start up another transfer in the EOT procedure. Note that you can not do a READ/READLN from the keyboard (since the kbd is interrupt driven and at level 1 and all the external interfaces are at level 3 and above). Be very careful in using this facility.


```

$COPYRIGHT 'COPYRIGHT (C) 1982 BY HEWLETT-PACKARD COMPANY'$
$SYSPROG ON$
$PARTIAL_EVAL ON$
$STACKCHECK ON$
$RANGE OFF$
$DEBUG OFF$
$OVFLCHECK OFF$

```

```

(*****
*)
*)      not released      VERSION      2.0      *)
*)
(*****
*)
*)      IOLIB              extensions      *)
*)
*)
(*****
*)
*)      library           - IOLIB          *)
*)      name              - EXTLIB        *)
*)      module(s)        - general_5      *)
*)
*)      date              - July 22 , 1982 *)
*)      update           - July 30 , 1982 *)
*)
*)
(*****
*)
*)
*)      GENERAL EXTENSIONS      *)
*)
*)
(*****

```

```

MODULE general_5 ;

    { date    07/22/82
      update  07/30/82

      purpose This module contains the LEVEL 5 GENERAL
              GROUP procedures.
    }

```

```

IMPORT iodeclarations ;

EXPORT

    TYPE user_eot_proc = PROCEDURE (parameter : INTEGER);

```

```

PROCEDURE on_eot      (VAR b_info: buf_info_type ;
                      your_proc : user_eot_proc ;
                      your_parm : INTEGER);
PROCEDURE off_eot    (VAR b_info: buf_info_type );

```

IMPLEMENT

```

PROCEDURE on_eot      (VAR b_info: buf_info_type ;
                      your_proc : user_eot_proc ;
                      your_parm : INTEGER);
TYPE proc_coerce     = RECORD CASE BOOLEAN OF
  TRUE: (user: PROCEDURE(parm:INTEGER));
  FALSE: (sys : PROCEDURE(parm:ANYPTR) )
END;
TYPE parm_coerce     = RECORD CASE BOOLEAN OF
  TRUE: (int : INTEGER);
  FALSE: (ptr : ANYPTR )
END;
VAR localproc : proc_coerce;
    localparm : parm_coerce;
BEGIN
  WITH b_info DO
    BEGIN
      localproc.user      := your_proc;
      eot_proc.real_proc := localproc.sys;
      localparm.int       := your_parm;
      eot_parm            := localparm.ptr;
    END; { of WITH DO }
  END; { of on_eot }

PROCEDURE off_eot    (VAR b_info: buf_info_type);
BEGIN
  WITH b_info DO
    BEGIN
      eot_proc.dummy_sl := NIL;
      eot_proc.dummy_pr := NIL;
      eot_parm          := NIL;
    END; { of WITH DO }
  END; { of on_eot }

END.    { of general_5 }

```

Interrupt Service Routine Procedures

Most of the drivers support a user-interrupt facility. When an user-interrupt mask has been enable, the condition has occurred and the driver ISR completes, the specified procedure is called. This is very similar to the ON INTR capability in BASIC. One major difference is that the procedure is called from inside an ISR (since that is where the condition was detected). There is no end-of-line in Pascal, so you have to be very careful of the operations inside an EOT procedure.

The current library has no high level support for user ISRs. Rather than force the use of a rather unpleasant mechanism, the following modules will provide a nice, high-level set of routines.

Note that you can do other I/O in the isr procedure. Note that you can not do a READ/READLN from the keyboard (since the kbd is interrupt driven and at level 1 and all the external interfaces are at level 3 and above). Be very careful in using this interrupt facility, it has not been thoroughly tested.

There are 3 modules for this interrupt facility -- HPIB_5, GPIO_5 and SERIAL_5. They are all very similar in structure. Each has a global variable (an array of pointers) that contains a pointer to NIL or to an allocated (heap) block of control information for the interrupt. So only the select codes that are enabled for interrupts will consume space for the control block.

Each interrupt condition has two procedures associated with it; enable and disable an interrupt on that condition. The general form is:

```
ON_condition (select_code , user_procedure , integer_parm);
```

```
OFF_condition (select_code , user_procedure , integer_parm);
```

Each condition is viewed as being separate from all other conditions (even on the same select code). So a user can have four different conditions on a particular interface handled by four separate procedures. Each user procedure must have a single INTEGER parameter -- even if it is not used. The intent of this parameter is open to the programmer. An example use of this parameter is when the programmer wishes to handle several interfaces in the same manner. The programmer can use the parameter to indicate the select code to his/her user procedure.

HP-IB Interrupts

The four interrupts for HP-IB are:

1. Talker
2. Listener
3. Controller
4. SRQ

An example use of HP-IB interrupts follows:

```
$SYSPROG ON$
PROGRAM isrtest(INPUT,OUTPUT);
$SEARCH '#3:HPIB5'$           { or wherever }
IMPORT iodeclarations,general_1,hpib_0,hpib_2,hpib_3,hpib_5;

VAR i      : INTEGER;

PROCEDURE myproc(temp : INTEGER);
BEGIN
  WRITELN('                ISR ');
  TRY
    i:=spoll(730);
    WRITELN('                ',i:4);
    clear_hpib(7,atn_line);    { so 98034 can re-assert srq line
                                since I used a 9835/98034 as a
                                device }
  RECOVER BEGIN
    WRITELN('                ISR ESCAPE');
    ioreset(7);
  END;
END;

BEGIN
  i:=-1;
  set_timeout(7,1.0);
  on_srq(7,myproc,0);
  WHILE TRUE DO BEGIN
    WRITELN('waiting ',i:4);
  END;
END.
```

```

$COPYRIGHT 'COPYRIGHT (C) 1982 BY HEWLETT-PACKARD COMPANY'$
$SYSPROG ON$
$PARTIAL_EVAL ON$
$STACKCHECK ON$
$RANGE OFF$
$DEBUG OFF$
$OVFLCHECK OFF$
(*****)
(*)
(*)      not released      VERSION      2.0      (*)
(*)
(*****)
(*)
(*)
(*)      IOLIB              extensions      (*)
(*)
(*)
(*****)
(*)
(*)
(*)      library           - IOLIB          (*)
(*)      name              - EXTLIB        (*)
(*)      module(s)         - hpib_5        (*)
(*)
(*)      date               - July 22 , 1982 (*)
(*)      update             - July 30 , 1982 (*)
(*)
(*)
(*****)

(*****)
(*)
(*)
(*)      GENERAL EXTENSIONS (*)
(*)
(*)
(*****)

```

```
PROGRAM hpib_5_init;
```

```
MODULE hpib_5 ;
```

```

{ date    07/22/82
  update  07/30/82

```

```

  purpose This module contains the LEVEL 5 HPIB GROUP
           procedures.

```

```
}
```

```
IMPORT iodeclarations , iocomasm , general_0 , hpib_1 , hpib_3 ;
```

```
EXPORT
```

```

TYPE hpib_user_proc = PROCEDURE (parameter : INTEGER);
TYPE hpib_isr_block = RECORD
    state : PACKED ARRAY[0..3] OF BOOLEAN;
    mask  : INTEGER;
    procs : ARRAY[0..3] OF hpib_user_proc;
    parms : ARRAY[0..3] OF INTEGER;
END;

VAR hpib_isr_table : ARRAY[iominisc..iomaxisc] OF
    ^hpib_isr_block;

PROCEDURE on_srq      (isc      : type_isc ;
                      your_proc : hpib_user_proc ;
                      your_parm : INTEGER);
PROCEDURE off_srq     (isc      : type_isc);

PROCEDURE on_talker  (isc      : type_isc ;
                      your_proc : hpib_user_proc ;
                      your_parm : INTEGER);
PROCEDURE off_talker (isc      : type_isc);

PROCEDURE on_listener (isc      : type_isc ;
                      your_proc : hpib_user_proc ;
                      your_parm : INTEGER);
PROCEDURE off_listener(isc      : type_isc);

PROCEDURE on_active_ctl
    (isc      : type_isc ;
     your_proc : hpib_user_proc ;
     your_parm : INTEGER);
PROCEDURE off_active_ctl
    (isc      : type_isc);

```

IMPLEMENT

```

CONST srqcond      = 0;      srqmask = 128;
      tlkcond      = 1;      tlkmask = 32;
      lstcond      = 2;      lstmask = 16;
      ctlcond      = 3;      ctlmask = 64;

TYPE coerce = RECORD CASE BOOLEAN OF
    TRUE: (int : INTEGER);
    FALSE: (ptr : ANYPTR)
END;

PROCEDURE hpib_isr_allocate
    (isc      : type_isc);
VAR counter : INTEGER;
BEGIN
    NEW(hpib_isr_table[isc]);
    WITH hpib_isr_table[isc]^ DO BEGIN
        FOR counter:=srqcond TO ctlcond DO state[counter] := FALSE;
        mask := 0;
    END; { of WITH DO BEGIN }
END; { of hpib_isr_allocate }

```

```

PROCEDURE hpib_isr_proc
    (temp          : ANYPTR );
VAR counter : INTEGER;
    happened: BOOLEAN;
    isc      : INTEGER;
    local    : coerce ;
BEGIN
    local_ptr := temp;                { coerce for select code }
    isc       := local.int;

    { prevent recursive hpib_isr_proc in user_isr }
    iocontrol(isc , 5 , 0);
    WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
        user_isr.dummy_sl := NIL;
        user_isr.dummy_pr := NIL;
    END; { of WITH isc_table DO BEGIN }

    WITH hpib_isr_table[isc]^ DO BEGIN
        FOR counter := srqcond TO ctlcond DO
            IF state[ counter ]
                THEN BEGIN
                    happened := FALSE;
                    CASE counter OF
                        srqcond: happened:=requested(isc);
                        tlkcond: happened:=talker(isc);
                        lstcond: happened:=listener(isc);
                        ctlcond: happened:=active_controller(isc);
                    END; { of CASE }
                    IF happened THEN CALL(procs[counter],parms[counter]);
                END; { of FOR DO IF bit_set THEN }

            { set up hpib_isr_proc in user_isr in temps }
            WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
                user_isr.real_proc := hpib_isr_proc;
            END; { of WITH DO BEGIN }

            { re - enable interrupts }
            iocontrol(isc , 5 , mask);

        END; { of WITH BEGIN }
    END; { of hpib_isr_proc }

PROCEDURE hpib_isr_setup
    (isc          : type_isc ;
     your_proc    : hpib_user_proc ;
     your_parm    : INTEGER ;
     which_cond   : INTEGER);
VAR local : coerce;
BEGIN
    IF (isc_table[isc].card_id <> hp98624) AND
        (isc_table[isc].card_id <> internal_hpib)
        THEN io_escape(ioe_not_hpib,isc);
    IF hpib_isr_table[isc] = NIL THEN hpib_isr_allocate(isc);
    WITH hpib_isr_table[isc]^ DO BEGIN
        { set up procedures & parameters in allocated isr proc block }
    
```

```

procs[which_cond] := your_proc;
parms[which_cond] := your_parm;

{ set up state condition and interrupt mask }
CASE which_cond OF
  srqcond: mask:=BINIOR(mask,srqmask);
  tlkcond: mask:=BINIOR(mask,tlkmask);
  lstcond: mask:=BINIOR(mask,lstmask);
  ctlcond: mask:=BINIOR(mask,ctlmask);
END; { of CASE }
state[which_cond] := TRUE;

{ set up hpib_isr_proc in user_isr in temps }
WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
  user_isr.real_proc := hpib_isr_proc;
  local.int           := isc;           { type coercion }
  user_parm           := local.ptr;     { type coercion }
END; { of WITH DO BEGIN }

{ enable card }
iocontrol(isc , 5 , mask);
END; { of WITH DO BEGIN }
END; { of hpib_isr_setup }

PROCEDURE hpib_isr_kill
      (isc           : type_isc;
       which_cond: INTEGER);
BEGIN
  IF hpib_isr_table[isc] <> NIL THEN
    WITH hpib_isr_table[isc]^ DO BEGIN

      { clear state condition and interrupt mask }
      CASE which_cond OF
        srqcond: mask:=BINAND(mask,BINCMP(srqmask));
        tlkcond: mask:=BINAND(mask,BINCMP(tlkmask));
        lstcond: mask:=BINAND(mask,BINCMP(lstmask));
        ctlcond: mask:=BINAND(mask,BINCMP(ctlmask));
      END; { of CASE }
      state[which_cond] := FALSE;

      { if necessary clear hpib_isr_proc in user_isr in temps }
      IF mask=0 THEN WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
        user_isr.dummy_sl := NIL;
        user_isr.dummy_pr := NIL;
        user_parm         := NIL;
      END; { of WITH isc_table DO BEGIN }

      { disable or enable card as specified by the mask }
      iocontrol(isc , 5 , mask);
    END; { of WITH DO BEGIN }
  END; { of hpib_isr_kill }

PROCEDURE on_srq      (isc           : type_isc ;
                      your_proc : hpib_user_proc ;
                      your_parm : INTEGER);

```



```

BEGIN
  hpib_isr_setup(isc,your_proc,your_parm,srqcond);
END;

PROCEDURE off_srq      (isc      : type_isc);
BEGIN
  hpib_isr_kill(isc,srqcond);
END;

PROCEDURE on_talker   (isc      : type_isc ;
                      your_proc : hpib_user_proc ;
                      your_parm : INTEGER);

BEGIN
  hpib_isr_setup(isc,your_proc,your_parm,tlkcond);
END;

PROCEDURE off_talker  (isc      : type_isc);
BEGIN
  hpib_isr_kill(isc,tlkcond);
END;

PROCEDURE on_listener (isc      : type_isc ;
                      your_proc : hpib_user_proc ;
                      your_parm : INTEGER);

BEGIN
  hpib_isr_setup(isc,your_proc,your_parm,lstcond);
END;

PROCEDURE off_listener(isc      : type_isc);
BEGIN
  hpib_isr_kill(isc,lstcond);
END;

PROCEDURE on_active_ctl
                      (isc      : type_isc ;
                      your_proc : hpib_user_proc ;
                      your_parm : INTEGER);

BEGIN
  hpib_isr_setup(isc,your_proc,your_parm,ctlcond);
END;

PROCEDURE off_active_ctl
                      (isc      : type_isc);
BEGIN
  hpib_isr_kill(isc,ctlcond);
END;

END; { of hpib_5 }

IMPORT iodeclarations , hpib_5;
VAR counter : INTEGER;
BEGIN
  FOR counter := iominisc TO iomaxisc DO
    hpib_isr_table[counter] := NIL;
  END. { of hpib_5_init }

```

GPIO Interrupts

There is one interrupt for GPIO which is the flag interrupt. An example use of GPIO interrupt follows:

```
$SYSPROG ON$
PROGRAM isrtest(INPUT,OUTPUT);
$SEARCH '#3:GPIO5'$ { or wherever }
IMPORT iodeclarations,general_1,gpio_5;

VAR i : INTEGER;

PROCEDURE myproc(temp : INTEGER);
BEGIN
  WRITELN('          ISR ');
  TRY
    readword(15,i);
    WRITELN('          ',i:6);
  RECOVER BEGIN
    WRITELN('          ISR ESCAPE');
  END;
END;

BEGIN
  i:=-1;
  set_timeout(15,1.0);
  on_flag(15,myproc,0);
  WHILE TRUE DO BEGIN
    WRITELN('waiting ',i:4);
  END;
END.
```

```

$COPYRIGHT 'COPYRIGHT (C) 1982 BY HEWLETT-PACKARD COMPANY'$
$SYSPROG ON$
$PARTIAL_EVAL ON$
$STACKCHECK ON$
$RANGE OFF$
$DEBUG OFF$
$OVFLCHECK OFF$
(*****
*)
*)      not released      VERSION      2.0      *)
*)
(*****
*)
*)      IOLIB              extensions      *)
*)
*)
(*****
*)
*)
*)      library           - IOLIB          *)
*)      name              - EXTLIB        *)
*)      module(s)         - gpio_5        *)
*)
*)      date               - July 22 , 1982 *)
*)      update            - July 30 , 1982 *)
*)
*)
(*****

(*****
*)
*)
*)      GENERAL EXTENSIONS      *)
*)
*)
(*****

```

```
PROGRAM gpio_5_init;
```

```
MODULE gpio_5 ;
```

```

{ date      07/26/82
  update    07/30/82

  purpose This module contains the LEVEL 5 GPIO GROUP
           procedures.
}
```

```
IMPORT iodeclarations , iocomasm , general_0 ;
```

```
EXPORT
```

```
TYPE gpio_user_proc = PROCEDURE (parameter : INTEGER);
```

```

TYPE gpio_isr_block = RECORD
    state : PACKED ARRAY[0..0] OF BOOLEAN;
    mask  : INTEGER;
    procs : ARRAY[0..0] OF gpio_user_proc;
    parms : ARRAY[0..0] OF INTEGER;
END;

VAR gpio_isr_table : ARRAY[iominisc..iomaxisc] OF
    ^gpio_isr_block;

PROCEDURE on_flag    (isc      : type_isc ;
                    your_proc : gpio_user_proc ;
                    your_parm : INTEGER);
PROCEDURE off_flag   (isc      : type_isc);

```

IMPLEMENT

```

CONST flgcond      = 0;      flgmask = 128;

TYPE coerce = RECORD CASE BOOLEAN OF
    TRUE: (int : INTEGER);
    FALSE: (ptr : ANYPTR)
END;

PROCEDURE gpio_isr_allocate
    (isc      : type_isc);
VAR counter : INTEGER;
BEGIN
    NEW(gpio_isr_table[isc]);
    WITH gpio_isr_table[isc]^ DO BEGIN
        FOR counter:=flgcond TO flgcond DO state[counter] := FALSE;
        mask := 0;
    END; { of WITH DO BEGIN }
END; { of gpio_isr_allocate }

PROCEDURE gpio_isr_proc
    (temp      : ANYPTR);
VAR counter : INTEGER;
    happened: BOOLEAN;
    isc      : INTEGER;
    local    : coerce ;
BEGIN
    local.ptr := temp;
    isc       := local.int;

    { prevent recursive gpio_isr_proc in user isr }
    iocontrol(isc , 5 , 0);
    WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
        user_isr.dummy_sl := NIL;
        user_isr.dummy_pr := NIL;
    END; { of WITH isc_table DO BEGIN }

    WITH gpio_isr_table[isc]^ DO BEGIN
        FOR counter := flgcond TO flgcond DO
            IF state[ counter ]

```

```

THEN BEGIN
    happened := FALSE;
    CASE counter OF
        flgcond: happened:=bit_set(ioread_byte(isc,0),0);
    END; { of CASE }
    IF happened THEN CALL(procs[counter],parms[counter]);
END; { of FOR DO IF bit_set THEN }

{ set up gpio_isr_proc in user_isr in temps }
WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
    user_isr.real_proc := gpio_isr_proc;
END; { of WITH DO BEGIN }

{ re - enable interrupts }
iocontrol(isc , 5 , mask);

END; { of WITH BEGIN }
END; { of gpio_isr_proc }

PROCEDURE gpio_isr_setup
    (isc          : type_isc ;
     your_proc    : gpio_user_proc ;
     your_parm    : INTEGER ;
     which_cond   : INTEGER);
VAR local : coerce ;
BEGIN
    IF (isc_table[isc].card_id <> hp98622)
        THEN io_escape(ioe_misc,isc);
    IF gpio_isr_table[isc] = NIL THEN gpio_isr_allocate(isc);
    WITH gpio_isr_table[isc]^ DO BEGIN
        { set up procedures & parameters in allocated isr proc block }
        procs[which_cond] := your_proc;
        parms[which_cond] := your_parm;

        { set up state condition and interrupt mask }
        CASE which_cond OF
            flgcond: mask:=BINIOR(mask,flgmask);
        END; { of CASE }
        state[which_cond] := TRUE;

        { set up gpio_isr_proc in user_isr in temps }
        WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
            user_isr.real_proc := gpio_isr_proc;
            local.int          := isc;                { type coercion }
            user_parm          := local.ptr;         { type coercion }
        END; { of WITH DO BEGIN }

        { enable card }
        iocontrol(isc , 5 , mask);
    END; { of WITH DO BEGIN }
END; { of gpio_isr_setup }

PROCEDURE gpio_isr_kill
    (isc          : type_isc ;
     which_cond   : INTEGER);
BEGIN

```

```

IF gpio_isr_table[isc] <> NIL THEN
WITH gpio_isr_table[isc]^ DO BEGIN

    { clear state condition and interrupt mask }
    CASE which_cond OF
        flgcond: mask:=BINAND(mask,BINCOMP(flmask));
    END; { of CASE }
    state[which_cond] := FALSE;

    { if necessary clear gpio_isr_proc in user_isr in temps }
    IF mask=0 THEN WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
        user_isr.dummy_sl := NIL;
        user_isr.dummy_pr := NIL;
        user_parm         := NIL;
    END; { of WITH isc_table DO BEGIN }

    { disable or enable card as specified by the mask }
    ioccontrol(isc , 5 , mask);
    END; { of WITH DO BEGIN }
END; { of gpio_isr_kill }

PROCEDURE on_flag      (isc      : type_isc ;
                       your_proc : gpio_user_proc ;
                       your_parm : INTEGER);
BEGIN
    gpio_isr_setup(isc,your_proc,your_parm,flgcond);
END;

PROCEDURE off_flag     (isc      : type_isc);
BEGIN
    gpio_isr_kill(isc,flgcond);
END;

END; { of gpio_5 }

IMPORT iodeclarations , gpio_5;
VAR counter : INTEGER;
BEGIN
    FOR counter := iominisc TO iomaxisc DO
        gpio_isr_table[counter] := NIL;
    END. { of gpio_5_init }

```

Serial Interrupts

The eight interrupts for the 98628 data comm card are:

1. Data Ready
2. Prompt Received
3. Frame or Parity Error
4. Modem Line Change
5. No Activity Timeout
6. Lost Carrier
7. End-of-Line Received
8. Break Received

An example use of data comm interrupts follows:

```

$SYSPROG ON$
PROGRAM isrtest(INPUT,OUTPUT);
$SEARCH '#3:SERIAL5'$           { or wherever }
IMPORT iodeclarations,general_0,general_1,general_2,
      serial_3,serial_5;

VAR i   : INTEGER;
      isc : INTEGER;

PROCEDURE myproc(temp : INTEGER);
BEGIN
  WRITELN('break received      ISR ');
END;

BEGIN
  isc:=-1;
  FOR i:=0 TO 31 DO IF isc_table[i].card_id=hp98628_async THEN isc:=i;
  WRITELN(isc);

  set_baud_rate   (isc,2400);
  set_parity      (isc,odd_parity);
  set_char_length (isc,7);
  set_stop_bits   (isc,1);

  iocontrol(isc,12,1);

  writestringln(isc,'ready when you are CB - to hit break');

  on_break(isc,myproc,0);

  i:=0;
  WHILE TRUE DO BEGIN
    i:=i+1;
    WRITELN('waiting ',i:6);
  END;
END.

```



```

$COPYRIGHT 'COPYRIGHT (C) 1982 BY HEWLETT-PACKARD COMPANY'$
$SYSPROG ON$
$PARTIAL_EVAL ON$
$STACKCHECK ON$
$RANGE OFF$
$DEBUG ON$
$OVFLCHECK OFF$

```

```

(*****)
(*)
(*) not released VERSION 2.0 (*)
(*)
(*****)
(*)
(*)
(*) IOLIB extensions (*)
(*)
(*)
(*****)
(*)
(*)
(*) library - IOLIB (*)
(*) name - EXTLIB (*)
(*) module(s) - serial_5 (*)
(*)
(*) date - July 22 , 1982 (*)
(*) update - July 30 , 1982 (*)
(*)
(*)
(*****)

(*****)
(*)
(*)
(*) GENERAL EXTENSIONS (*)
(*)
(*)
(*****)

```

```
PROGRAM serial_5_init;
```

```
MODULE serial_5 ;
```

```

{ date 07/26/82
  update 07/30/82

  purpose This module contains the LEVEL 5 SERIAL GROUP
           procedures.
}

```

```
IMPORT iodeclarations , iocomasm , general_0 ;
```

```
EXPORT
```

```
TYPE serial_user_proc = PROCEDURE (parameter : INTEGER);
```

```

TYPE serial_isr_block = RECORD
    state : PACKED ARRAY[0..7] OF BOOLEAN;
    mask  : INTEGER;
    procs : ARRAY[0..7] OF serial_user_proc;
    parms : ARRAY[0..7] OF INTEGER;
END;

VAR serial_isr_table : ARRAY[iominisc..iomaxisc] OF
    ^serial_isr_block;

PROCEDURE on_data      (isc      : type_isc ;
                       your_proc : serial_user_proc ;
                       your_parm : INTEGER);
PROCEDURE off_data     (isc      : type_isc);

PROCEDURE on_prompt   (isc      : type_isc ;
                       your_proc : serial_user_proc ;
                       your_parm : INTEGER);
PROCEDURE off_prompt  (isc      : type_isc);

PROCEDURE on_fp_error (isc      : type_isc ;
                       your_proc : serial_user_proc ;
                       your_parm : INTEGER);
PROCEDURE off_fp_error(isc      : type_isc);

PROCEDURE on_modem    (isc      : type_isc ;
                       your_proc : serial_user_proc ;
                       your_parm : INTEGER);
PROCEDURE off_modem   (isc      : type_isc);

PROCEDURE on_no_activity
    (isc      : type_isc ;
     your_proc : serial_user_proc ;
     your_parm : INTEGER);
PROCEDURE off_no_activity
    (isc      : type_isc);

PROCEDURE on_lost_carrier
    (isc      : type_isc ;
     your_proc : serial_user_proc ;
     your_parm : INTEGER);
PROCEDURE off_lost_carrier
    (isc      : type_isc);

PROCEDURE on_eol      (isc      : type_isc ;
                       your_proc : serial_user_proc ;
                       your_parm : INTEGER);
PROCEDURE off_eol     (isc      : type_isc);

PROCEDURE on_break    (isc      : type_isc ;
                       your_proc : serial_user_proc ;
                       your_parm : INTEGER);
PROCEDURE off_break   (isc      : type_isc);

```

IMPLEMENT

```

CONST data_cond    = 0;   data_mask = 1;   { data ready  }
      prmpt_cond   = 1;   prmpt_mask = 2;   { prompt     }
      fperr_cond   = 2;   fperr_mask = 4;   { frame/parity }
      mdmch_cond   = 3;   mdmch_mask = 8;   { modem change }
      noact_cond   = 4;   noact_mask = 16;  { no activity  }
      lstcr_cond   = 5;   lstcr_mask = 32;  { lost carrier }
      eol_cond     = 6;   eol_mask  = 64;  { end of line  }
      break_cond   = 7;   break_mask = 128; { break       }

```

```

TYPE coerce = RECORD CASE BOOLEAN OF
    TRUE: (int : INTEGER);
    FALSE: (ptr : ANYPTR)
END;

```

```

PROCEDURE serial_enable
    (isc      : type_isc ;
     newmask  : INTEGER);

```

```

VAR x : INTEGER;
BEGIN

```

```

    { There are two interrupt mask areas - the general card
      interrupt mask and the ON INTR interrupt facility within the
      card's interrupts. The iocontrol register 13 is the ON INTR
      mask. The drv_misc[3] AND iocontrol register 121 is the
      general card interrupt mask. }

```

```

    WITH isc_table[ isc ].io_tmp_ptr^ DO BEGIN
        iocontrol (isc , 13+256 , newmask); { set ON INTR mask }
        x := ORD(drv_misc[3]);             { get usr0mask   }
        IF newmask = 0 THEN x := BINAND(x,BINCOMP(8))
        ELSE x := BINIOR(x,8);
        drv_misc[3] := CHR(x);              { set/clr bit 3 in }
                                           {   usr0mask       }
        iocontrol (isc , 121+256 , x);      { set/clr bit 3 in }
                                           {   ctl reg 121   }

```

```

    END; { of WITH DO BEGIN }
END; { of serial_enable }

```

```

PROCEDURE serial_isr_allocate
    (isc      : type_isc);

```

```

VAR counter : INTEGER;
BEGIN
    NEW(serial_isr_table[isc]);
    WITH serial_isr_table[isc]^ DO BEGIN
        FOR counter:=data_cond TO break_cond
            DO state[counter] := FALSE;
        mask := 0;
    END; { of WITH DO BEGIN }
END; { of serial_isr_allocate }

```

```

PROCEDURE serial_isr_proc
    (temp      : ANYPTR );

```

```

VAR counter : INTEGER;
    happened: BOOLEAN;
    isc      : INTEGER;

```

```

        local      : coerce ;
        reason     : INTEGER;
BEGIN
    local_ptr := temp;                { coerce to get sc }
    isc       := local.int;

    reason := iostatus (isc , 4);

    { prevent recursive serial_isr_proc in user_isr }
    serial_enable(isc , 0);
    WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
        user_isr.dummy_sl := NIL;
        user_isr.dummy_pr := NIL;
    END; { of WITH isc_table DO BEGIN }

    WITH serial_isr_table[isc]^ DO BEGIN
        FOR counter := data_cond TO break_cond DO
            IF state[ counter ]
                THEN BEGIN
                    happened := bit_set(reason , counter);
                    IF happened THEN CALL(procs[counter],parms[counter]);
                END; { of FOR DO IF bit_set THEN }

        { set up serial_isr_proc in user_isr in temps }
        WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
            user_isr.real_proc := serial_isr_proc;
        END; { of WITH DO BEGIN }

        { re - enable interrupts }
        serial_enable(isc , mask);

    END; { of WITH BEGIN }
END; { of serial_isr_proc }

PROCEDURE serial_isr_setup
    (isc          : type_isc ;
     your_proc    : serial_user_proc ;
     your_parm    : INTEGER ;
     which__cond : INTEGER);

VAR local : coerce;
BEGIN
    IF (isc_table[isc].card_id <> hp98628_async) AND
        (isc_table[isc].card_id <> hp_datacomm)
        THEN io_escape(ioe_misc,isc);
    IF serial_isr_table[isc] = NIL THEN serial_isr_allocate(isc);
    WITH serial_isr_table[isc]^ DO BEGIN
        { set up procedures & parameters in allocated isr proc block }
        procs[which__cond] := your_proc;
        parms[which__cond] := your_parm;

        { set up state _condition and interrupt mask }
        CASE which__cond OF
            data_cond:   mask:=BINIOR(mask,data_mask );
            prmpt_cond:  mask:=BINIOR(mask,prmpt_mask);
            fperr_cond:  mask:=BINIOR(mask,fperr_mask);
            mdmch_cond:  mask:=BINIOR(mask,mdmch_mask);
        END CASE;
    END;
END;

```

```

        noact_cond:  mask:=BINIOR(mask,noact_mask);
        lstcr_cond:  mask:=BINIOR(mask,lstcr_mask);
        eol_cond:    mask:=BINIOR(mask,eol_mask  );
        break_cond:  mask:=BINIOR(mask,break_mask);
END; { of CASE }
state[which__cond] := TRUE;

{ set up serial_isr_proc in user_isr in temps }
WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
    user_isr.real_proc := serial_isr_proc;
    local.int          := isc;                { type coerce }
    user_parm          := local.ptr;         { type coerce }
END; { of WITH DO BEGIN }

{ enable card }
serial_enable(isc , mask);
END; { of WITH DO BEGIN }
END; { of serial_isr_setup }

PROCEDURE serial_isr_kill
    (isc          : type_isc ;
     which__cond: INTEGER);
BEGIN
    IF serial_isr_table[isc] <> NIL THEN
        WITH serial_isr_table[isc]^ DO BEGIN

            { clear state condition and interrupt mask }
            CASE which__cond OF
                data_cond:  mask:=BINAND(mask,BINCOMP(data_mask ));
                prmpt_cond: mask:=BINAND(mask,BINCOMP(prmpt_mask));
                fperr_cond: mask:=BINAND(mask,BINCOMP(fperr_mask));
                mdmch_cond: mask:=BINAND(mask,BINCOMP(mdmch_mask));
                noact_cond: mask:=BINAND(mask,BINCOMP(noact_mask));
                lstcr_cond: mask:=BINAND(mask,BINCOMP(lstcr_mask));
                eol_cond:   mask:=BINAND(mask,BINCOMP(eol_mask  ));
                break_cond: mask:=BINAND(mask,BINCOMP(break_mask));
            END; { of CASE }
            state[which__cond] := FALSE;

            { if necessary clear serial_isr_proc in user_isr in temps }
            IF mask=0 THEN WITH isc_table[isc].io_tmp_ptr^ DO BEGIN
                user_isr.dummy_sl := NIL;
                user_isr.dummy_pr := NIL;
                user_parm         := NIL;
            END; { of WITH isc_table DO BEGIN }

            { disable or enable card as specified by the _mask }
            serial_enable(isc , mask);
        END; { of WITH DO BEGIN }
    END; { of serial_isr_kill }

PROCEDURE on_data    (isc          : type_isc ;
                     your_proc   : serial_user_proc ;
                     your_parm   : INTEGER);
BEGIN

```

```

    serial_isr_setup(isc,your_proc,your_parm,data_cond);
END;
PROCEDURE off_data      (isc          : type_isc);
BEGIN
    serial_isr_kill(isc,data_cond);
END;

PROCEDURE on_prompt    (isc          : type_isc ;
                        your_proc    : serial_user_proc ;
                        your_parm    : INTEGER);

BEGIN
    serial_isr_setup(isc,your_proc,your_parm,prmt_cond);
END;
PROCEDURE off_prompt   (isc          : type_isc);
BEGIN
    serial_isr_kill(isc,prmt_cond);
END;

PROCEDURE on_fp_error  (isc          : type_isc ;
                        your_proc    : serial_user_proc ;
                        your_parm    : INTEGER);

BEGIN
    serial_isr_setup(isc,your_proc,your_parm,fperr_cond);
END;
PROCEDURE off_fp_error(isc          : type_isc);
BEGIN
    serial_isr_kill(isc,fperr_cond);
END;

PROCEDURE on_modem     (isc          : type_isc ;
                        your_proc    : serial_user_proc ;
                        your_parm    : INTEGER);

BEGIN
    serial_isr_setup(isc,your_proc,your_parm,mdmch_cond);
END;
PROCEDURE off_modem   (isc          : type_isc);
BEGIN
    serial_isr_kill(isc,mdmch_cond);
END;

PROCEDURE on_no_activity
                        (isc          : type_isc ;
                        your_proc    : serial_user_proc ;
                        your_parm    : INTEGER);

BEGIN
    serial_isr_setup(isc,your_proc,your_parm,noact_cond);
END;
PROCEDURE off_no_activity
                        (isc          : type_isc);

BEGIN
    serial_isr_kill(isc,noact_cond);
END;

PROCEDURE on_lost_carrier
                        (isc          : type_isc ;
                        your_proc    : serial_user_proc ;

```

```

                                your_parm : INTEGER);
BEGIN
    serial_isr_setup(isc,your_proc,your_parm,lstcr_cond);
END;
PROCEDURE off_lost_carrier
                                (isc          : type_isc);
BEGIN
    serial_isr_kill(isc,lstcr_cond);
END;

PROCEDURE on_eol                (isc          : type_isc ;
                                your_proc    : serial_user_proc ;
                                your_parm    : INTEGER);
BEGIN
    serial_isr_setup(isc,your_proc,your_parm,eol_cond);
END;
PROCEDURE off_eol              (isc          : type_isc);
BEGIN
    serial_isr_kill(isc,eol_cond);
END;

PROCEDURE on_break             (isc          : type_isc ;
                                your_proc    : serial_user_proc ;
                                your_parm    : INTEGER);
BEGIN
    serial_isr_setup(isc,your_proc,your_parm,break_cond);
END;
PROCEDURE off_break            (isc          : type_isc);
BEGIN
    serial_isr_kill(isc,break_cond);
END;

END; { of serial_5 }

IMPORT iodeclarations , serial_5;
VAR counter : INTEGER;
BEGIN
    FOR counter := iominisc TO iomaxisc DO
        serial_isr_table[counter] := NIL;
    END.    { of serial_5_init }

```

Chapter 20

The DIO Bus

Introduction

The Series 200 Desktop-computer Input/Output (DIO) Bus standard defines both mechanical and electrical requirements of cards which are to be used as optional input/output (I/O) devices with HP Series 200 Computers. The DIO Bus was first implemented in the Model 26 Computer (HP 9826A), followed shortly thereafter by the Model 36 (9836A) and the Model 16 (9816A). The 9888A Bus Expander, which can be used with Series 200 Computers, also implements the DIO Bus.

The DIO Bus is designed around the MC68000 series of microprocessors. If you want further information regarding MC68000 operation, refer to the *MC68000 User's Manual*, HP part number 09826-90073.

Objectives

The purpose of this document is to provide sufficient documentation to permit experienced digital-hardware designers to develop devices for use with the DIO Bus. The goal is to provide enough information to design Bus Slaves -- in particular, I/O cards. Bus Masters, such as Processor and DMA Controller boards, cannot be designed by using the information in this document. You may want to use the HP 98630 Breadboard Interface as the beginning of your own custom interface.

Any questions you may have regarding the information in this document should be brought to the attention of your local HP Desktop Computer Systems Engineer.

Designer's Responsibilities

In order to ensure safe, reliable operation with Series 200 Computer products, the specifications in this document must be strictly followed when designing Bus Slave devices. The "Electrical Specifications" and "Mechanical Specifications" sections describe topics such as available power-supply current and size requirements of I/O cards. The section called "Design Qualification" provides safety and operating requirements that your I/O card design must meet to qualify as a usable device.

Keep in mind that you are responsible for any circuitry that you design and use with HP products, both in terms of personal safety and proper operation with the equipment.

CAUTION

HEWLETT-PACKARD SHALL NOT BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL. REPAIRS NECESSITATED BY MISUSE OF THE EQUIPMENT, OR BY HARDWARE, SOFTWARE, OR INTERFACING NOT PROVIDED BY HEWLETT-PACKARD ARE NOT COVERED BY THE WARRANTY.

Signal Terminology

The following convention is used throughout this document: Active-low signals are denoted with a * following the name. This is equivalent to a bar over the signal name which is often used for active-low signals. Thus, the following are equivalent:

$$\text{BAS*} = \overline{\text{BAS}}$$

When a signal is referenced as "asserted" or "true," "negated" or "false," and so forth, it is relative to the signal's *function*. For example, to say that BAS is asserted means that it is active (performing its function). Whether it exists as BAS (active-high) or BAS* (active-low) on the backplane is irrelevant.

References to "high" and "low" refer directly to TTL logic voltage levels. When referring to high and low signals, the actual name of the signal is used. For example, when the signal BAS* is described as being low, the signal entitled BAS* has a TTL logic-low voltage level. The TTL logic levels are defined as follows:

Logic High: ≥ 2.0 volts

Logic Low: ≤ 0.8 volts

System Elements

The functional modules of the DIO Bus are shown below. Where signals go specifically from one functional module to another, the two modules are shown side-by-side for clarity.

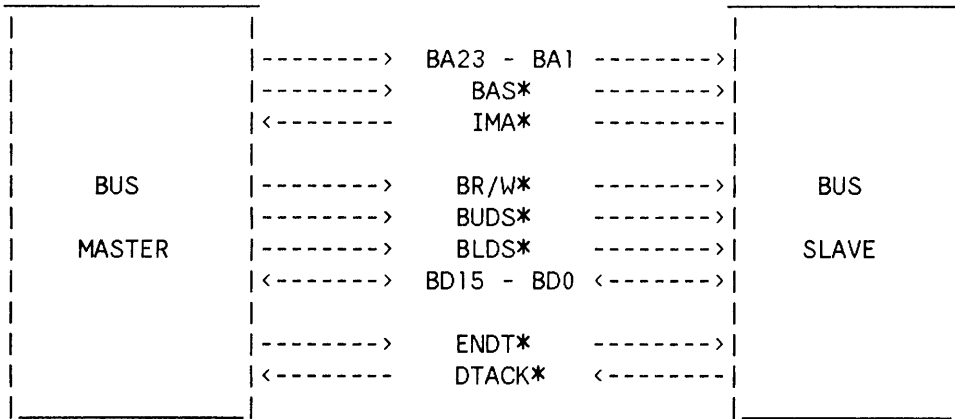


Figure 1a. Data-Transfer System Elements

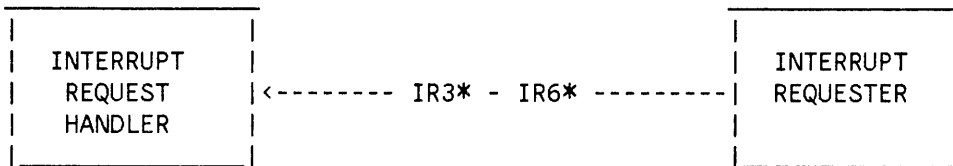


Figure 1b. Interrupt System Elements

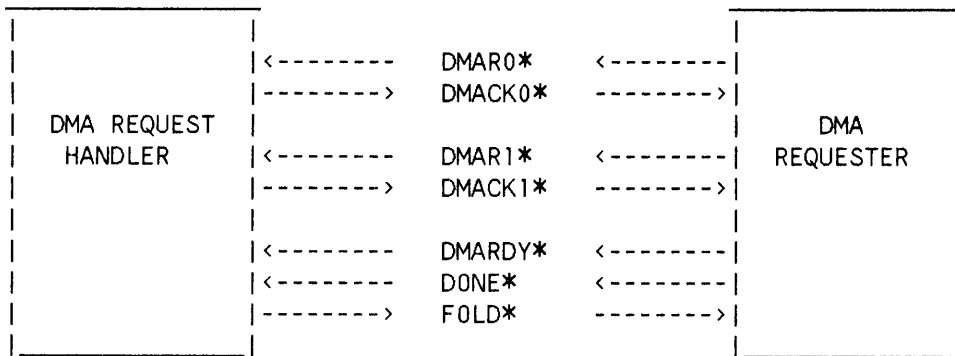
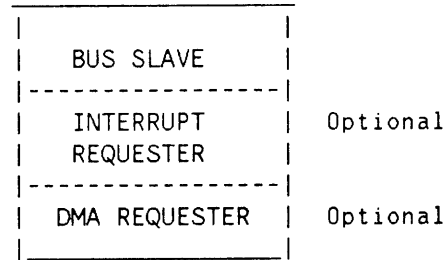


Figure 1c. Direct-Memory Access System Elements

Bus Slave Subsystem

A diagram of a Bus Slave subsystem is shown in the following drawing.

BUS SLAVE SUBSYSTEM



The Bus Slave subsystem consists of the system interface elements listed on the right hand side of Figures 1a, 1b, and 1c.

Bus Timing Background

The key feature of the DIO Bus is that it is *asynchronous* -- in other words, there is no clock signal on the backplane to which other signals are referenced. While address and data generation are related to the CPU clock, the actual clock does not appear on the bus. The presence of address or data is indicated by various control lines which execute interlocked handshakes to convey address and data. Because the address, data, and control lines are not referenced to a clock on the backplane, *signal skew* must be controlled to maintain the relative timing between these signals.

For example, the MC68000 microprocessor is guaranteed to drive the bus address lines 30 ns prior to asserting Address Strobe. Most receiving devices require at least 15 ns of address setup time prior to Address Strobe. To guarantee 15 ns of address setup time, the following rules were developed to control gate delays and bus loading (these are expanded in greater detail in later sections).

- Each board is limited to one LS TTL load on the address bus, data bus, the address strobe, the data strobes, and read/write signal.
- The PC board trace length on bus signals should be as short as possible and, in any case, must not exceed 3 inches.
- An SN74LS245 (or equivalent SN74LS244) is used to buffer the above signals.

Thus the designer is given the guideline that, from the input of the 74LS245 bus driver to the input of the slave's bus receivers, 15 ns of skew are possible (see Figure 2). Signal skews are due to *differences* in device delays and physical properties of bus lines (such as capacitance). Therefore, skews due to the bus drivers and the bus are not specified separately. Detailed signal-loading specifications are discussed in the "Electrical Specifications" section.

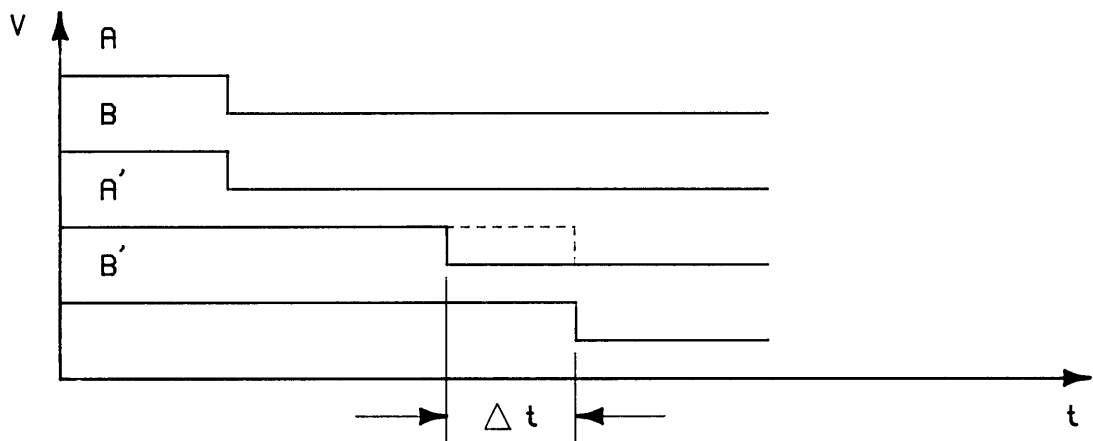
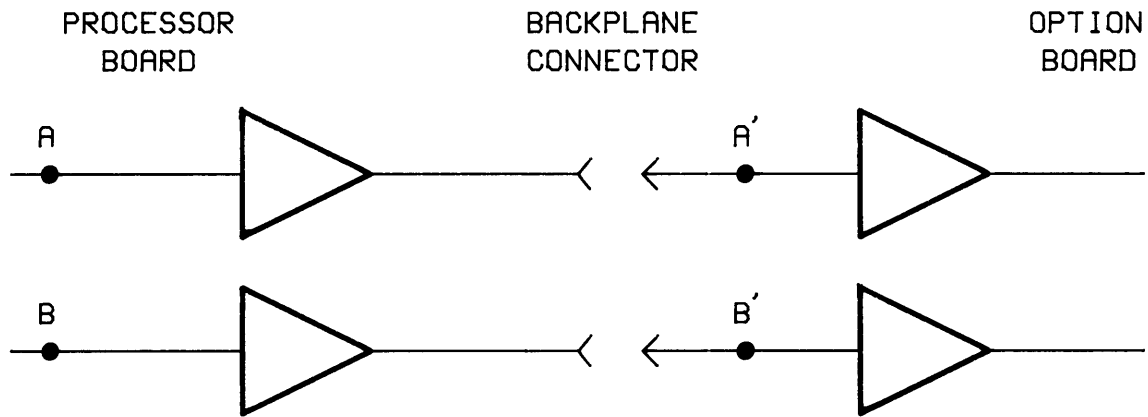


Figure 2. Origins of Signal Skew

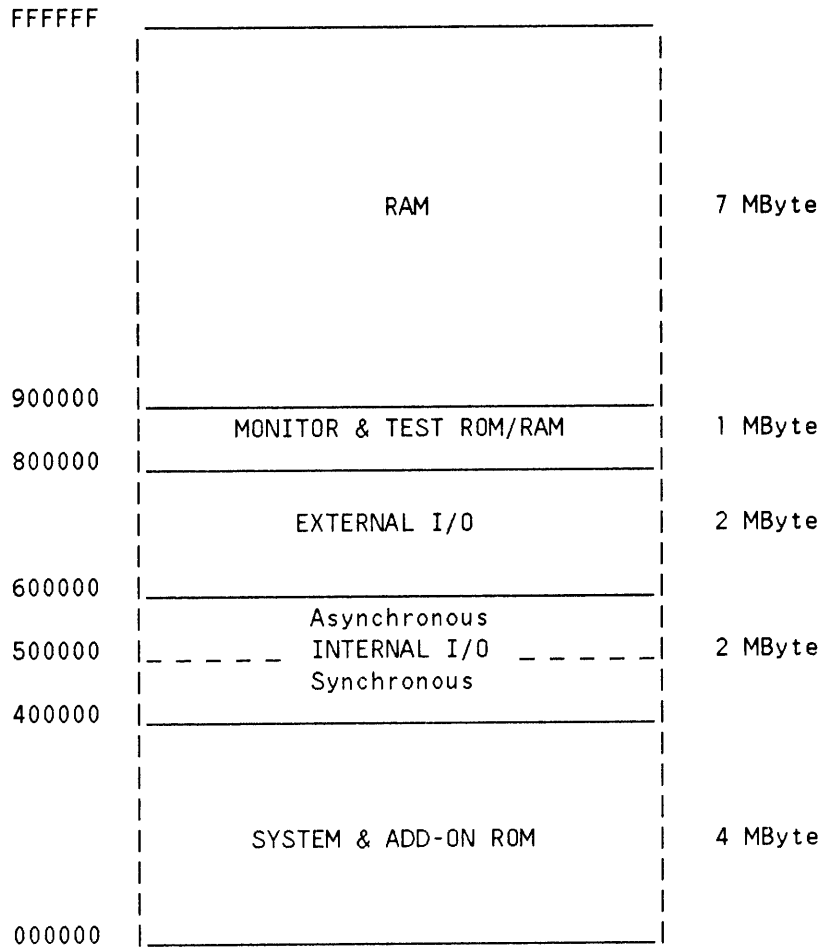
Memory Map

This Bus specification is not intended to document in detail all Series 200 Computers' memory maps. Instead, documentation of the memory map is limited to the External I/O memory map and standard I/O register assignments.

Series 200 Memory Map

The Series 200 memory map is shown below. The 68000's 24-bit address bus can directly address 16 Mbytes of memory. The External I/O occupies 2 Mbytes of address space (hexadecimal addresses 600000 through 7FFFFF).

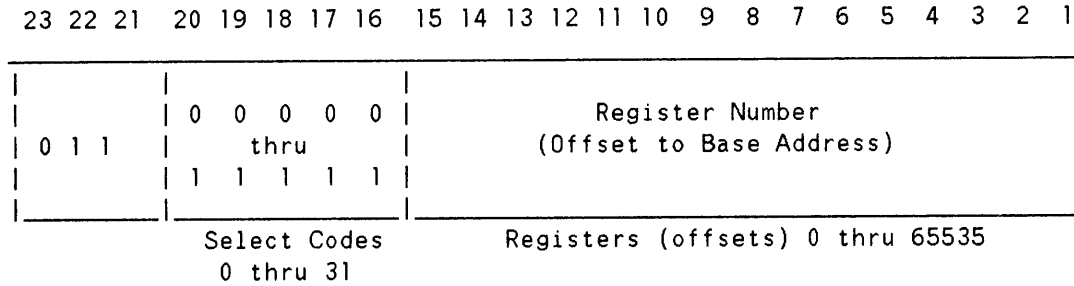
Hex Address



External I/O Memory Map

The External I/O address space is divided into 32 segments of 64 Kbytes each. The I/O cards contain select code switches which determine the physical address of the card in the External I/O address space. Five switches permit the user to choose one of 32 select codes, ranging from 0 through 31, to determine which 64 Kbyte memory space the card resides in. Switches should be implemented in the I/O card design for flexibility reasons. The address format, shown below, locates I/O devices in memory locations 600 000 through 7FF FFF. Note that all registers or memory locations on an I/O card are offsets to the card's "base address."

Address Bit:



	SELECT CODE	BASE ADDRESS	STANDARD ASSIGNMENT	
7FFFFF	31	7F0000	Reserved	
	30	7E0000	Reserved	
	29	7D0000	98627 (continued)	
	28	7C0000	98627 Color Output	
	27	7B0000	Reserved	
	26	7A0000	Reserved	
	25	790000	Reserved	
	24	780000	Reserved	
	23	770000	Reserved	
	22	760000	Reserved	
	21	750000	98629A SRM	
	20	740000	98628A Datacomm	
	19	730000	Reserved	
	18	720000	Reserved	
	17	710000	Reserved	
	16	700000	Custom I/O Card 2	
	15	6F0000	Custom I/O Card 1	
	14	6E0000	98625 Disc	
	13	6D0000	Reserved	
	12	6C0000	98622 GPIO	
	11	6B0000	98623 BCD	
	10	6A0000	Reserved	
	9	690000	98626 RS-232	Note 1
	8	680000	98624 Ext. HP-IB	
	7	670000		
	6	660000		
	5	650000		
	4	640000		
	3	630000		
	2	620000		
	1	610000		
600000	0	600000		

NOTE 1. The 98626A interface built into the 9816A is "hardwired" to this Select Code.

Registers

The function of certain registers within I/O cards are pre-assigned. Note that because most I/O cards are byte-oriented and these registers are connected to the lower byte of the data bus, their memory addresses are odd (1, 3, 5, and so forth) relative to the card's base address.

The designer is free to implement registers in addition to (but not instead of) the ones listed below. Also, the designer is not required to uniquely map each register to a location within the card's address space (i.e. several offset addresses may access the same register, which simplifies address decoding, as long as the addresses are not outside the card's 64 Kbyte address space).

Standard I/O Registers

The standard I/O card registers are defined as follows; the register number is the offset (added to the base address of the card) which is used to access the register.

Read Register 1: ID Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	ID4	ID3	ID2	ID1	ID0
Value =128	Value =64	Value =32	Value =16	Value =8	Value =4	Value =2	Value =1

ID4 thru ID0 -- Contain the card ID, which uniquely identifies each type of I/O card.

Currently Defined ID Numbers

0 - Reserved	16 - Custom I/O Card 2
1 - 98624	17 - Reserved
2 - 98626	18 - Reserved
3 - 98622	19 - Reserved
4 - 98623	20 - 98628A/98629A
5 - Reserved	21 - Reserved
6 - Reserved	22 - Reserved
7 - Reserved	23 - Reserved
8 - 98625	24 - Reserved
9 - Reserved	25 - Reserved
10 - Reserved	26 - Reserved
11 - Reserved	27 - Reserved
12 - Reserved	28 - 98627
13 - Reserved	29 - Reserved
14 - Reserved	30 - Reserved
15 - Custom I/O Card 1	31 - Reserved

Note that two ID numbers, 15 and 16, have been reserved for custom I/O cards designed and implemented outside of Hewlett-Packard.

Write Register 1: Interface Reset

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
The value written into this register is irrelevant.							
Value =128	Value =64	Value =32	Value =16	Value =8	Value =4	Value =2	Value =1

Writing any value into this register performs an **Interface Reset** of the card. The card's actual response to this action depends on how it is designed.

Good system design requires that the operating system should be capable of resetting an I/O card to its power-on state. One of two methods must be implemented:

1. If the card contains LSI chips, one or more commands may be defined which can be sent to gracefully return the card to its power-on state.
2. If the card does not have such a sequence, the card may be capable of being reset to its power-on state by writing to register 1.

Read Register 3: Interrupt and DMA Status

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IE	IR	INT LVL Switches		Undefined		DE1	DE0
Value =128	Value =64	Value =32	Value =16	Value =8	Value =4	Value =2	Value =1

IE -- Interrupts Enabled: If this bit is set, interrupts are enabled.

IR -- Interrupt Request: If this bit is set, the card is requesting an interrupt. This bit is used during software polling to determine interrupt origin.

INT LVL Switches -- Interrupt Level: The interrupt level is typically set by two switches on the I/O card; these switches map into the 2 Interrupt Level bits.

Switch Setting	Interrupt Level
0 0	3
0 1	4
1 0	5
1 1	6

Undefined -- These bits have no standard definition and are thus available for user-defined functions.

DE1 -- When this bit is set, DMA is enabled on channel 1.

DE0 -- When this bit is set, DMA is enabled on channel 0.

Write Register 3: Interrupt and DMA Enable

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
EI	Undefined					DE1	DE0
Value =128	Value =64	Value =32	Value =16	Value =8	Value =4	Value =2	Value =1

EI -- Setting this bit enables interrupts.

Undefined -- These bits have no standard definition and are thus available for user-defined functions.

DE1 -- Setting this bit enables DMA operations on Channel 1.

DE0 -- Setting this bit enables DMA operations on Channel 0.

Data Transfers

This section discusses the transfer of data between Bus Masters and Bus Slaves. Briefly stated, data transfers on the DIO Bus are made by writing data to a memory location (or reading data from a location). The Bus Master (the CPU, or Processor, board) defines the address, data direction, and whether a byte or word is to be transferred. The Bus Slave device that contains the memory location makes an acknowledgement to the master, which completes the transfer.

The rest of this section describes the signals involved in the transfer and the transfer process.

Data Transfer Signals

The bus signals used in data transfers are shown below. Signal names starting with B (for "buffered") are derived from signal names of the 68000 microprocessor -- the 68000 name is that which follows the "B" in the name. A brief description of each signal is given; for more detailed information on the buffered 68000 signals, refer to the *MC68000 User's Manual*, HP part number 09826-90073. Two of the signals, IMA* and ENDT*, are HP-defined.

BA23--BA1 The 23-bit address bus. Note that BA0 is not on the bus; its meaning is conveyed in BUDS* and BLDS* (see below).

BAS* Buffered Address Strobe, defines when the address is valid.

BD15--BD0 The 16-bit data bus.

BR/W* Buffered Read/Write: High for read, Low for write.

BUDS*, Buffered Upper Data Strobe, Buffered Lower Data **BLDS***, Strobe and Buffered Data Strobe: BUDS* indicates BDS* that BD15-BD8 are required; BLDS* indicates that BD7-BD0 are required. BDS* is used generically to refer to either BLDS* or BUDS*. It is not a bus signal; it is used for discussion purposes only.

DTACK* Data Transfer Acknowledge: issued by the currently addressed slave (RAM, I/O card, etc.) to inform the master that the slave can complete the transfer cycle. During a read operation, it indicates that data signals placed on the bus by the slave are now valid. During a write operation, it indicates that the slave has accepted the data.

IMA* I'm Addressed: issued by a card that detects itself being addressed. It is also used by the Bus Expander to reverse its data-bus buffers when a card in the Bus Expander is addressed.

ENDT* Enable DTACK: generated by the CPU board and (optionally) used by Bus Slaves to generate the DTACK* signal automatically, which reduces the access-overhead time and provides a pseudo-synchronous, repeatable access-cycle time. Note that this signal does not have to be implemented in a Bus Slave design.

Data Transfer Overview

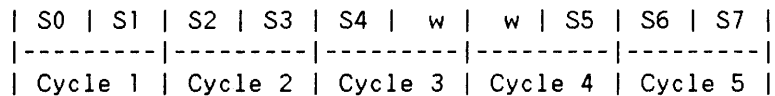
Before the transfer is initiated, the R/W* line is used to indicate the direction of data flow. BAS*, which defines when the bus address is valid, begins the data transfer operation. As soon as BAS* is asserted, each Bus Slave should check to see if the address currently on the address bus is contained within its address space. Once a Bus Slave detects that it is being addressed, it must acknowledge with IMA*. BLDS* and BUDS* indicate which data lines are involved in the transfer.

An interlocked handshake is used to transfer data from the Bus Master to the Bus Slave. The steps of the handshake are as follows:

1. Bus Slave asserts DTACK* when data accepted from bus or provided on bus.
2. Bus Master sees DTACK* and enters "terminate cycle" sequence.
3. Bus Master negates BAS* when cycle completes.
4. Bus Slave negates DTACK* when BAS* negated.

As mentioned earlier, ENDT* is available from the Bus Master. It is used to improve the response time of Bus Slaves. ENDT* is basically BAS* delayed by one and one-half clock cycles (of an 8 MHz clock). A Bus Slave can use this signal to pseudo-synchronize an asynchronous process (e.g. dynamic memory) so that access time is fixed (at five clock cycles). ENDT* timing is discussed later in this chapter.

The reference to clock cycles should not be confused with the state designators S0 through S7 shown on 68000 timing diagrams. One clock cycle is equal to two states. For instance, one clock cycle corresponds to states S0 and S1 on the 68000 timing diagrams. The following diagram shows the relationship of the CPU-board clock cycles and 68000 state designators.



Read Cycle Description

Figure 3 shows the timing of signals during a read cycle. Timing specifications are for Bus Slaves. The key aspects of a read cycle are as follows:

1. Prior to the beginning of the read cycle, BR/W*, BUDS*, BLDS* and DTACK* are actively pulled high. The setup time on BR/W* high is 15 ns before BAS* goes low. Because this is a read cycle, BR/W* remains high during the entire cycle.
2. The Bus Master drives the address bus BA23-BA1 with a minimum address setup time of 15 ns before BAS* is asserted.
3. All Bus Slaves determine if they are being addressed using BAS* as a decode enable; the device being addressed responds with IMA* within 50 ns after BAS* occurs.
4. When a Bus Slave is addressed, it puts data on the bus after BDS* is asserted. (Note that BDS* can precede or follow BAS* by up to 75 ns.) DTACK* is asserted by the Bus Slave to indicate that the data signals are valid and that the transfer can be completed by the Bus

Master reading the data; thus, the time from BDS* to data valid is device-dependent. Bus Slaves must drive the data bus 30 ns prior to asserting DTACK* (except when using ENDT* or during memory-to-I/O DMA transfer). This 30 ns set-up time requirement must be met *at the receiving device*, so it must include all signal skews (see specification 9a of Figure 3).

Note

The MC68000 specification for asynchronous data setup time permits valid data to follow DTACK* by as much as 75 ns, due to delays in synchronizing DTACK*; however, to support DMA and other Bus Masters, the DIO Bus requires data to precede DTACK* (except when using ENDT*; see specification 9 of Figure 3).

For memory-to-I/O DMA transfers, the data must be valid 45 ns prior to DTACK*. This is to accommodate delay through the Fold Buffer (on the HP 98620 DMA controller card) as well as any additional bus delay. Additional bus delay occurs because twice the normal bus load is driven (the bus load *to* the Fold Buffer and the bus load *from* the Fold Buffer to the I/O card). Up to 30 ns of delay can be permitted and still meet the write timing specification of 15 ns data setup time prior to DTACK* (see specification 9b of Figure 3). The DMA section further describes DMA timing.

For I/O-to-memory DMA transfers, the data setup time from the I/O card is still specified at 30 ns. This is because there are matching delays in both the data (Fold Buffer plus 2 times normal bus loading) and the DTACK* signal (I/O card generates DMARDY* which the DMA Controller delays in "converting" to DTACK*). The DMA section further describes DMA timing.

5. The Bus Master detects that DTACK* has occurred and ends the cycle by negating BAS* and BDS*. BAS* is set false within 350 ns of assertion of DTACK*.
6. The Bus Slave detects that the cycle has ended when BAS* or BDS* go false (whichever occurs first) and then stops driving IMA*, DTACK*, and the Data Bus. Likewise, the Bus Master stops driving the address bus. Relative to BAS* or BDS* (whichever occurs last), the following signals change with the indicated delay:

Address hold: 15 ns (min.)
IMA* high: 50 ns (max.)
DTACK* high: 50 ns (max.)
Data Bus hold: 100 ns (max.)

Note

Processor Boards actively drive DTACK* high when BAS* goes high after a read or write cycle. Because the I/O card will keep DTACK* low until it sees BAS* high, there is a brief (<100 ns) DTACK* driver conflict until the I/O card stops driving DTACK*.

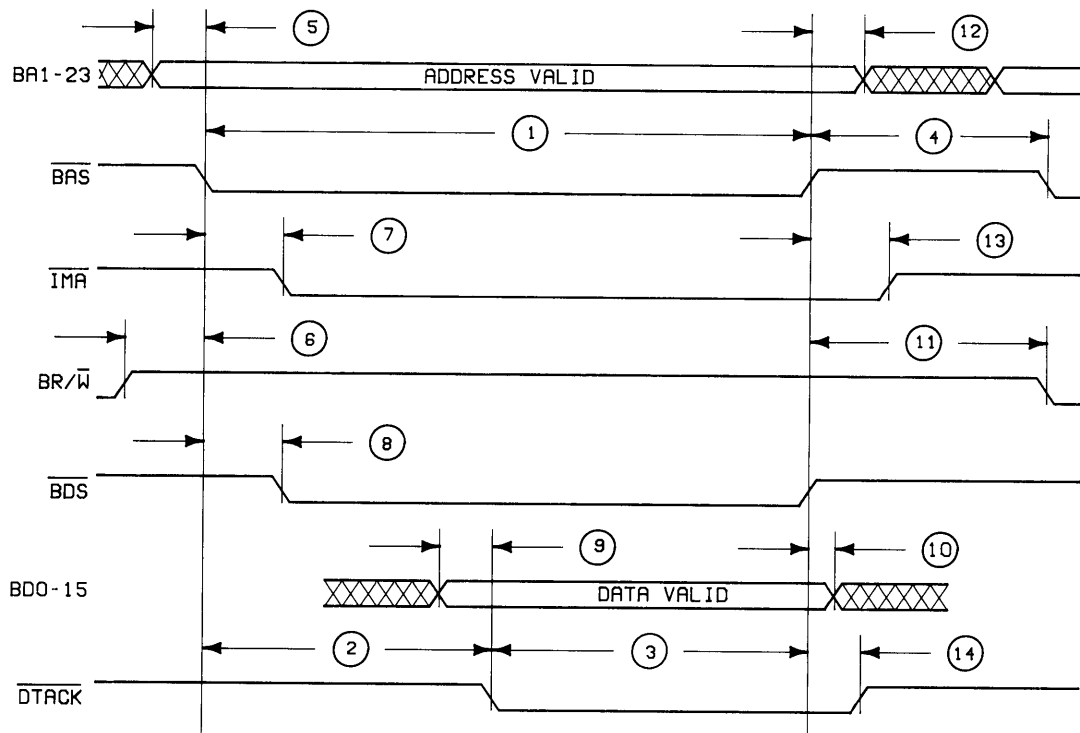


Figure 3. Read Cycle Timing

Read Cycle Times		Min.	Max.	
1	BAS* low (limited by BERR* on CPU board)		4500	
2	BAS* low to DTACK* low (without Bus Error):			
a)	DMA Operation		1500	
b)	Non-DMA Operation		3000	
3	DTACK* low to BAS* high (without Bus Error)		350	Note 1
4	BAS* high	140		
5	Address setup before BAS* low	15		
6	BR/W* high to BAS* low or BDS* low	15		
7	BAS* low to IMA* low	0	50	
8	BAS* low to BDS* low	-75	75	
9	Data setup before DTACK* low:			
a)	Data transfers without ENDT* or I/O-to-memory DMA transfers	30		
b)	Memory-to-I/O DMA transfers	45		
c)	Data transfers with ENDT*	-75		
10	Data hold after BAS* or BDS* high	0	100	Note 2
11	BR/W* high after BAS* or BDS* high	15		Note 3
12	Address hold after BAS* or BDS*	15		Notes 3&4
13	BAS* high to IMA* high	0	50	
14	BAS* high to DTACK* high	0	50	

Notes

1. This time must be met if Time 2 is 3000 ns. Otherwise, the requirement is that Time 2 + Time 3 < 3500 ns.
2. Whichever goes high first of BAS* or BDS*.
3. Whichever goes high last of BAS* or BDS*.
4. Based on the 68000 and buffer/bus skew specifications, the address hold time is 15 ns.

Write Cycle Description

Figure 4 shows the timing of signals during a write cycle. Timing specifications are for Bus Slaves. The key aspects of a write cycle are as follows:

1. Prior to the beginning of the write cycle, BR/W*, BUDS*, BLDS* and DTACK* are actively pulled high by the Bus Master.
2. Address is valid on the address bus BA23-BA1 with a minimum address set-up time of 15 ns before BAS* is asserted. The address is valid a minimum of 5 ns before BR/W* goes low (Write). BR/W* goes low sometime in the interval from 75 ns before to 85 ns after the assertion of BAS*.
3. All devices on the bus determine if they are being addressed using BAS as a decode enable; the device being addressed responds with IMA* within 50 ns after BAS* occurs.

4. Data is valid on the data bus BD15-BD0 a minimum of 15 ns prior to the assertion of BDS*. Notice that the time from BAS* to the data strobes can vary over a wide range (50 ns to 2500 ns). Longer times can occur when doing a DMA operation, because a read must be performed prior to the write operation.
5. When BDS* is asserted (low), BR/W* is guaranteed to already be asserted (low); BR/W* should *not* be qualified with BAS* because of the potential for bus conflicts, since BR/W* can still be changing up to 85 ns after BAS* goes true.
6. The Bus Slave stores the data and asserts DTACK* indicating to the Bus Master that the storage operation is complete.
7. The Bus Master detects that DTACK* is true and negates BAS* and BDS* within 350 ns. The Bus Master then removes the data BD15-BD0 from the data bus. To ensure data hold time sufficient to allow the Bus Slave to clock the data at the same time it asserts DTACK*, the minimum Bus Master data hold time after detection of DTACK* is 85 ns.
8. The following signals change with the indicated delay after the negation of BDS* and BAS* (whichever occurs last).

Address hold: 15 ns (min.)

BR/W* high: 25 ns (min.)

DMA* high: 50 ns (max.)

DTACK* high: 50 ns (max.)

DTACK* (a pull-up 0 ns (min.) is asserted on the Processor Board)

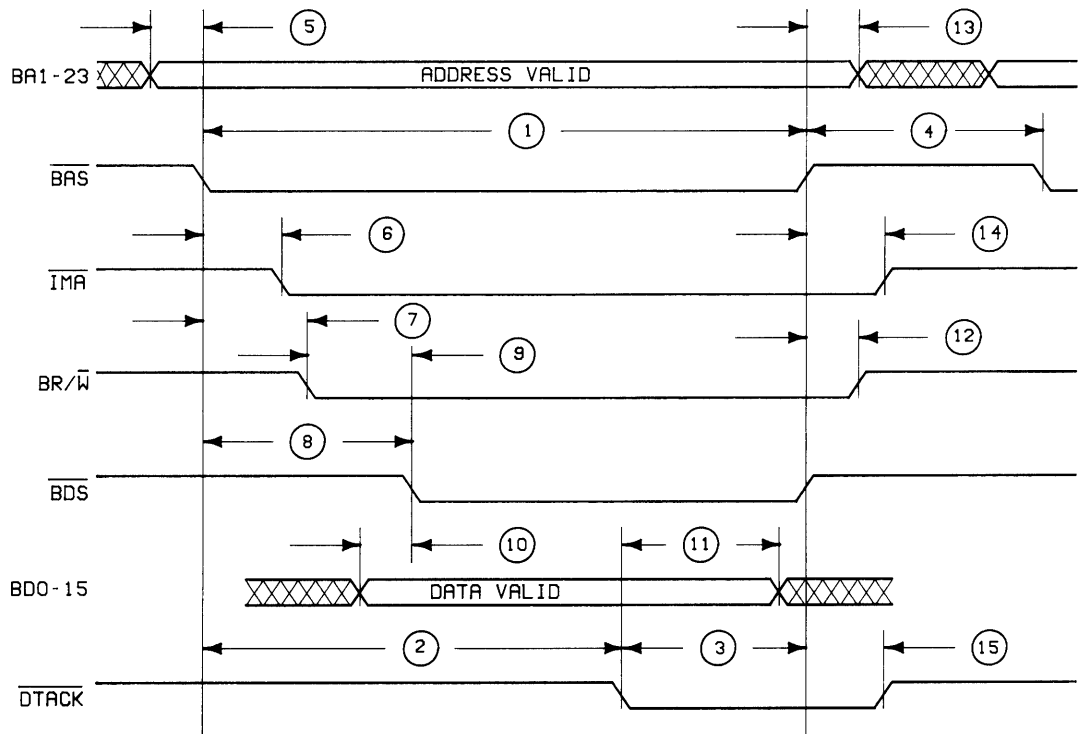


Figure 4. Write Cycle Timing

Write Cycle Times		Min.	Max.	
1	BAS* low (limited by BERR* on CPU board)		4500	
2	BAS* low to DTACK* low (without Bus Error)		3000	
3	DTACK* low to BAS* high (without Bus Error)		350	Note 1
4	BAS* high	140		
5	Address setup before BAS*	15		
6	BAS* low to IMA* low	0	50	
7	BAS* low to BR/W* low	-75	85	
8	BAS* low to BDS* low	50	2500	
9	BR/W* low to BDS* low	65		Note 2
10	Data setup before BDS* low	15		
11	Data hold after DTACK* low	85		
12	BR/W* hold after BAS* or BDS* high	15		Note 3
13	Address hold after BAS* or BDS*	15		Note 4
14	BAS* high to IMA* high	0	50	
15	BAS* high to DTACK* high	0	50	

Notes

1. This time must be met if Time 2 is 3000 ns. Otherwise, the requirement is that Time 2 + Time 3 < 3500 ns.
2. Times 7 & 8 imply that BR/W* could go low after BDS* goes low; in actuality, this cannot happen as guaranteed by Time 9.
3. Whichever of BAS* or BDS* goes high last.
4. Based on the 68000 and buffer/bus skew specifications, the address hold time is 15 ns.

Read-Modify-Write

No current I/O cards support read-modify-write operations.

Enable DTACK Timing

As discussed previously, Enable DTACK (ENDT*) is used to improve the response time of backplane cards by generating DTACK* pseudo-synchronously, permitting memory accesses to always occur in five clock cycles (of an 8 MHz clock). ENDT* is currently used only by Series 200 RAM cards. However, I/O cards that contain large amounts of memory can use ENDT* to speed up the transfer of data to and from this memory, since the access time of memory is usually fixed. During a read, DTACK* is generated before data is set up on the bus; however, because of synchronization delays, the 68000 permits data to follow DTACK*.

ENDT* is generated by the Processor board and is delayed from BAS* by one and one-half clock cycles, as shown below. The Bus Slave must have a 25 ns "turnaround" time from the assertion of ENDT* to asserting DTACK*. Also, data must be valid within 265 ns of BAS* being asserted during a read cycle.

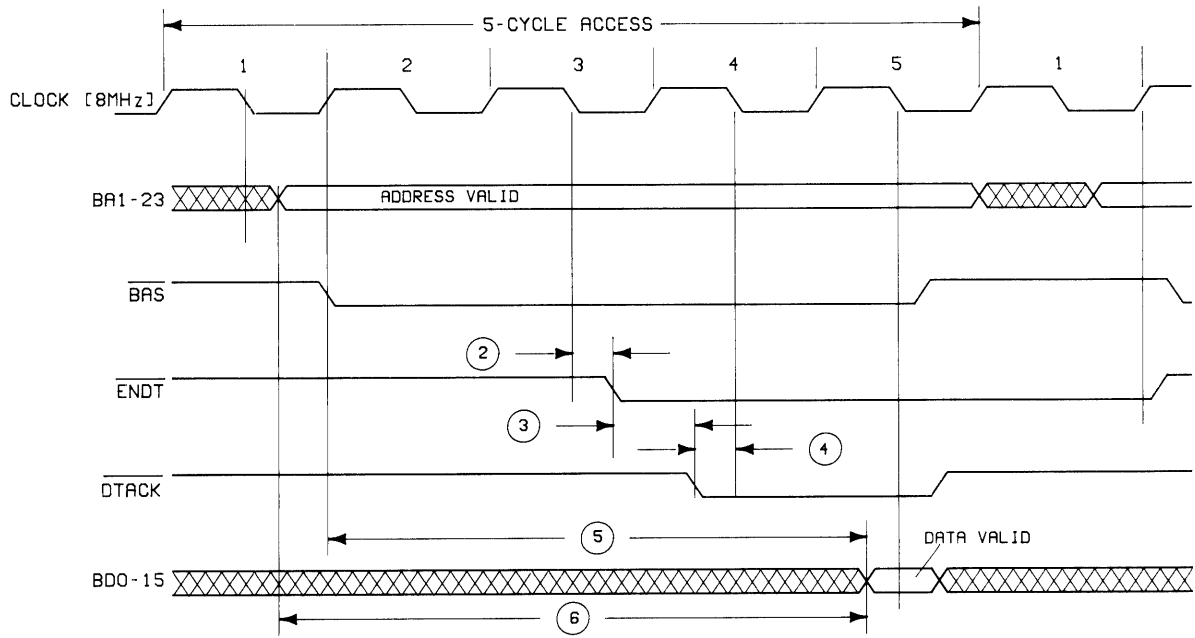


Figure 5. Enable DTACK Timing

ENDT* Timing		Min.	Max.	
1	Clock cycle time @ 8 Mhz	125 ns	nominal	Note 1
2	Clock low to ENDT* low		40	
3	ENDT* low to DTACK* low		50	Note 2
4	DTACK* setup time		35	
5	BAS* low to read data valid		265	
6	Address to read data valid		295	Note 3

Notes

1. The clock is on the Processor board and does not appear on the DIO Bus; it is shown for reference only.
2. To ensure that DTACK* is low at the Processor board input within 50 ns, the maximum ENDT* to DTACK* gate delay on the Bus Slave device *cannot* exceed 25 ns; this must represent the worse-case gate delay. The DTACK* driver must ensure that DTACK* is driven low on the bus within an additional 25 ns.
3. Whichever of these times is longer.

Direct Memory Access

All DMA operations with Series 200 Computers require the use of the HP 98620 Direct Memory Access (DMA) Controller card. This card monitors DMA requests from I/O cards, requests control of the bus, and orchestrates DMA data transfers. The DIO Bus supports two direct memory access channels. DMA transfers between I/O cards and memory are supported; memory-to-memory transfers are not. DMA data rates exceeding 1 million transfers/second are possible in word (or byte) mode. This section gives an overview of DMA operation and discusses DMA input and output operation.

DMA Signals

The signals unique to DMA operation are listed below. In addition to these signals, the normal Master/Slave data transfer signals are used (see the previous chapter).

DMAR0*, DMA Request: asserted by an I/O card to request **DMAR1*** a DMA transfer on DMA Channel 0 or Channel 1.

DMACK0* DMA Acknowledge: a response from the DMA **DMACK1*** Controller which acknowledges DMA request on Channel 0 or Channel 1.

DMARDY* DMA Ready: indicates that the I/O card has provided the data (DMA input) or accepted the data (DMA output).

DONE* Done: an output from the DMA Controller to indicate that DMA is done. **DONE*** can be used at the option of the I/O card designer to determine when DMA is done.

FOLD* Fold: an output from the DMA Controller to indicate that a data byte is being folded from the upper byte of the data bus to the lower byte (or vice versa). This folding is performed by the DMA Controller.

In the discussions that follow, **DMAR0*** and **DMACK0*** are used; however, all operations apply equally to **DMAR1*** and **DMACK1***.

DMA Overview

To enable a DMA transfer, the operating system (or program) must perform two operations:

1. Program the DMA Controller with the type of transfer (word/byte, input/output, priority, etc.).
2. Enable DMA channel 0 or DMA channel 1 on the I/O card by writing to Write Register 3. The I/O card will then request a DMA operation on the assigned channel. In response to this request, the DMA Controller requests and eventually receives control of the bus and then begins the DMA transfer.

A DMA transfer occurs during a single bus cycle during which data is both read and stored. For a DMA output cycle, the data is fetched from memory and written to the I/O card. For a DMA input cycle, the data is read from the I/O card and stored in memory. The I/O card itself is programmed to request the DMA transfer; upon seeing this DMA request, the DMA Controller

requests and receives control of the bus and provides the necessary address and control signals for the transfer.

During a DMA operation, the memory device does a normal data transfer using BAS^* , BR/W^* , BDS^* , $DTACK^*$, etc. Therefore, the I/O card must use different signals to handshake data. As discussed above, the I/O card asserts $DMAR0^*$ to request a DMA transfer. Once the DMA Controller has control of the bus, it responds with $DMACK0^*$ which the I/O card treats the same as BAS^* in that it begins the actual data transfer cycle. When the I/O card has provided or accepted the data, it responds with $DMARDY^*$, which the DMA Controller interprets as $DTACK^*$ and responds accordingly.

Both byte and word DMA transfers are supported. In word mode, data is transferred a word at a time between memory and the I/O device. In byte mode, data is transferred on the lower byte of the I/O card; however, because the data in memory is "packed", both upper and lower bytes of memory must be accessed. The DMA Controller supports this via a "Fold Buffer," which is used to transfer data between the upper byte of the data bus (for memory accesses) and the lower byte of the data bus for I/O cards. During a DMA input operation, the Fold Buffer is alternately used to transfer I/O data to the upper byte of memory (BD15-BD8). Likewise, during a DMA output operation, the Fold Buffer is used to transfer memory data on BD15-BD8 to the lower data byte for the I/O card. The DMA Controller provides the $FOLD^*$ signal indicating when folding is to occur; this is used primarily by the Bus Expander. Bus Slave designs should not incorporate this signal.

Note that, depending on programming of the DMA Controller, the speed of the I/O card and the speed of the peripheral, the DMA Controller may give up control of the bus between DMA cycles. Relinquishing of bus control depends upon two factors: 1) The time for the I/O card to request another DMA transfer and, 2) the channel priority programmed into the DMA Controller. This subject is discussed in more detail below.

DMA Output Cycle Description

Figure 6 shows the DMA Output cycle. To do a DMA output, a memory read is followed by an I/O write. Again, DMA Channel 0 is assumed; all operations apply equally to DMA Channel 1.

1. The I/O card asserts $DMAR0$, indicating that it is ready to begin a DMA output operation.
2. The DMA Controller detects this request and, if not the current Bus Master, it requests, and is eventually granted, the system bus.
3. The DMA Controller then initiates what looks like a normal memory read cycle:
 - a. Memory address is put on the bus and BR/W^* line is set to Read (high).
 - b. BAS^* and BDS^* are asserted for the memory device and DMA Acknowledge ($DMACK0^*$) is asserted to indicate to the I/O card that a DMA cycle has started. The I/O card responds to $DMACK0^*$ as it does to BAS^* during a normal transfer. If the DMA transfer is a word transfer, $BLDS^*$ and $BUDS^*$ are strobed simultaneously. If a byte transfer, $BLDS^*$ or $BUDS$ is strobed (depending on the byte being read). If the upper byte is being read, the Fold Buffer is used to transfer data from the upper byte to the lower byte of the data bus for the I/O card.
4. When the I/O card detects $DMACK0$, it can optionally release $DMAR0$. This is discussed in more detail below.

5. The memory device fetches the data, places it on the bus with 30 ns of setup time and asserts DTACK*. The I/O card detects DTACK* and, reacting to it like it normally reacts to BDS*, begins its own sequence to accept the data. If priority is not set for channel 0 and DMAR0* was set false after DMACK0*, the I/O card must re-assert DMAR0* to request the next cycle. In either case, the I/O card asserts DMARDY* when it has accepted the data.
6. The DMA Controller detects that DMARDY* has occurred and, responding to it like Bus Masters normally respond to DTACK*, ends the cycle by removing BAS*, BDS*, and DMACK0*. Once the I/O card detects that DMACK0* is gone, it removes DMARDY*.

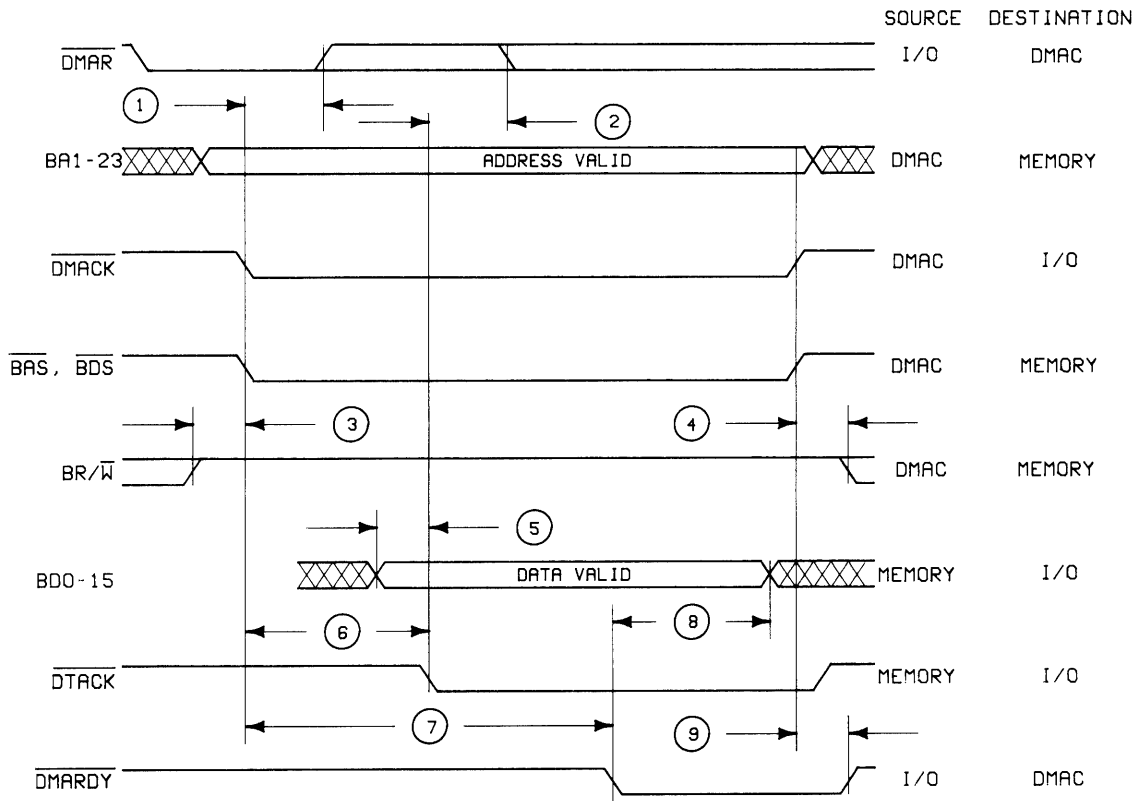


Figure 6. DMA Output Cycle Timing

DMA Output Timing		Min.	Max.	
1	DMAR* release after DMACK* low	0		Note 1
2	DMAR* low after DTACK* low:			
a)	Priority = 0		65	
b)	Priority = 1		1600	
3	BR/W* setup before DMACK* low	15		
4	BR/W* hold after DMACK* high	15		
5	Data setup before DTACK* low	15		
6	BAS* low to DTACK* low	0	1500	
7	DMACK* low to DMARDY* low (w/o Bus Error)		3000	
8	Data hold after DMARDY* low	85		
9	DMARDY* release after DMACK* high	0	50	

Notes

1. The I/O card may keep DMAR* low after its request is acknowledged.

DMA Input Cycle Description

Figure 7 shows the DMA input cycle. To do a DMA input, an I/O read is followed by a memory write. Again, DMA Channel 0 is assumed; all operations apply equally to DMA Channel 1.

1. The I/O card asserts DMAR0, indicating that it is ready to begin a DMA input operation.
2. The DMA Controller detects this request and, if not the current Bus Master, it requests, and is eventually granted, the system bus.
3. The DMA Controller then initiates what looks like a normal memory write cycle:
 - a. Memory address is put on the bus and BR/W* line is set to write (low). Notice that BR/W* is set low prior to BAS contrary to a normal write cycle in which BR/W* may go low after BAS. Since the DMA Controller knows that a memory write operation is to occur, it can assert BR/W immediately.
 - b. BAS* is asserted for the memory device and DMACK0* is asserted to indicate to the I/O card that a DMA cycle has started.
4. The I/O card responds to DMACK0* the same as it does to BAS* during a non-DMA transfer in that it enables the data transfer. When the I/O card detects DMACK0*, it can optionally release DMAR0*; this is discussed in more detail below. In response to DMACK0*, the I/O card fetches the data, places it on the bus and, after a minimum data setup time of 30 ns, asserts DMARDY* to indicate to the DMA Controller that the bus data is valid.
5. If the DMA transfer is a byte transfer and the data is to be written to the upper byte of memory, the DMA Controller uses its Fold Buffer to move the byte from the lower data byte to the upper data byte. In either case, the DMA Controller detects that DMARDY* has occurred and asserts the BLDS* and/or BUDS* to indicate to memory that data is valid on the bus.

6. The memory then stores the data and asserts DTACK* to indicate that data has been accepted. The DMA Controller detects that DTACK* has occurred and ends the cycle by removing BAS*, BDS*, and DMACK0*. In response to the removal of BAS*, the memory card removes DTACK*; likewise, in response to the removal of DMACK0*, the I/O card removes DMARDY*.
7. On the last byte, the DMA Controller generates the DONE* signal to tell the I/O card that "this is the last byte." An I/O card can, at its option, use this control line to inhibit further DMA Requests. Once the transfer count is satisfied, the DMA controller ignores further DMA Requests and relinquishes the bus.
8. A bus error also causes the DMA Controller to terminate the DMA transfer and relinquish the bus. A bus error occurs if a DMARDY* does not occur within 2.5 us of DMACK0* going true.

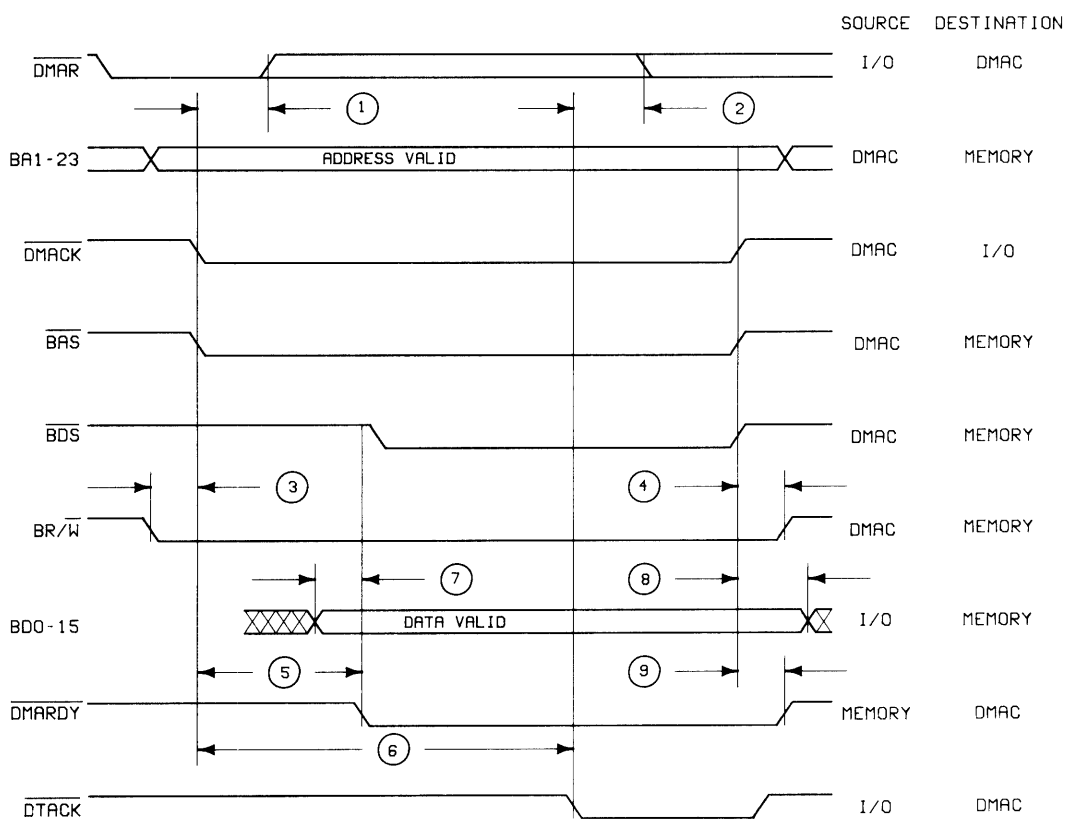


Figure 7. DMA Input Cycle Timing

DMA Input Cycle Timing		Min.	Max.
1	DMAR* release after DMACK* low	0	
2	DMAR* low after DTACK* low:		
a)	Priority = 0		65
b)	Priority = 1		1600
3	BR/W* low before DMACK* low	15	
4	BR/W* high after DMACK* high	15	
5	DMACK* low to DMARDY* low (w/o Bus Error)		2500
6	BAS* low to DTACK* low (w/o Bus Error)		3000
7	Data setup before DMARDY* low	30	
8	Data hold after DMACK* high	0	100
9	DMARDY* release after DMACK* high	0	50

DMA Speed Considerations

To optimize the speed of DMA transfers, the transfer (read/write) must be completed during a single bus cycle. Also, the overhead time of the DMA Controller must be minimal and the device connected to the I/O card must be able to provide or accept the data immediately. The time for the existing DMA Controller (98620A) to synchronize the handshake signals is similar to the response of the 68000 and adds minimal overhead.

To meet the desired performance, the DMA Controller must also minimize overhead time between bus cycles. The DMA Controller is designed to hold the bus continuously providing that the I/O card can generate another DMA Request (DMAR0*) within a certain length of time after DTACK*.

The amount of time that the I/O card has after DTACK* depends on the current setting of the Priority bit on the DMA Controller card. This time is either 65 ns (DMA Priority bit = 0) or 1.6 μ s (Priority bit = 1). See specification 2 in Figures 6 and 7. Priority bit set to 0 requires a 65 ns response; if the I/O card does not assert DMAR0* within 65 ns, the DMA Controller relinquishes the bus at the end of the cycle to the next highest priority bus master (typically the Master Controller).

Because designing an I/O card to respond to DTACK* with DMAR0* within 65 ns adds complexity to the I/O card, the DMA Controller can be programmed for the 1.6 μ s DTACK*-to-DMAR0* response time in the hope that the I/O card will generate another DMA Request. As mentioned above, selection of this time is done with the Priority bit, which operates as follows:

- If the Priority bit is 0 the I/O card must assert DMAR0* within 65 ns of DTACK*. This ensures that the DMA Controller keeps control of the bus and provides 1.2 Mbytes/sec.
- If the Priority signal for the channel is 1, then the bus is not relinquished until 1.6 μ s after the last transfer is complete.

Terminating DMA Transfers

DMA transfers can be terminated in several ways:

1. The DMA Controller can be programmed to interrupt the Bus Master after the transfer is complete and the bus relinquished.
2. The Bus Master can monitor the ARM bit in the DMA Controller between DMA cycles (if the bus is released) and when the DMA operation is complete. When ARM is 0, the transfer is complete.
3. The I/O card can use the DONE* signal from the DMA Controller to interrupt the Bus Master.

The DONE* signal works as follows: DONE* is asserted by the DMA Controller on the last byte of a DMA transfer. For an input operation, DONE* can be used by the I/O card to inhibit further acceptance or handshaking of data from the peripheral. Without the DONE* signal, the I/O card, not realizing that the transfer is complete, could accept the next byte from the peripheral. This can result in data being lost (unless the transfer count is set to the actual size minus 1).

For an output operation, the DONE* signal is not typically needed, since the DMA Controller simply ignores DMAR0* from the I/O card when the transfer count is satisfied. If desired, the DONE* signal can be used by the I/O card to inhibit an extra DMA Request. DONE* has sufficient setup and hold time to be qualified off of DMACK*.

Bus Error Operation

An exception sequence is generated when the Processor Board's Bus Error (BERR*) signal is asserted. This is an open-collector signal, permitting it to be generated by any device (including the CPU board). Current applications of the Bus Error signal are discussed in this chapter. This section is for informational purposes; the Bus Error line should not be implemented with Bus Slaves. Refer to the *MC68000 User's Manual* for timing requirements of the 68000 processor.

The Bus Error Signal

One signal is involved in bus error operation: the Bus Error signal (BERR*). When asserted, this signal causes the 68000 processor to terminate the current bus cycle and float the address and data bus. When negated, the processor begins its exception processing.

Bus Timeouts

The Master Controller will generate BERR* when an accessed device fails to respond within a certain time. As discussed previously, a device responds with DTACK*. Thus, the time from asserting BAS* to the arrival of DTACK* is monitored; since DTACK* causes BAS* to go false (with some delay), the BERR* circuit simply monitors the length of BAS*. If DTACK* occurs too late or fails to occur, BAS* will remain true and a counter will time out.

A four-bit BERR* counter, located on Series 200 CPU boards, remains cleared as long as BAS* is false. When BAS* is true, it begins counting (4 Mhz) and generates a BERR* when it overflows after 16 counts ($16 \times 250 \text{ ns} = 4.0 \mu\text{s}$). To provide margin, the maximum width of BAS* is set at $3.5 \mu\text{s}$. To ensure that DTACK* has time to negate BAS* prior to $3.5 \mu\text{s}$, DTACK* must arrive two CPU clocks cycles (250 ns) earlier. Because of this time and other delays, and to provide ample margin, the maximum time from BAS* going low to DTACK* going low has been specified at $3.0 \mu\text{s}$.

From the above description, it appears that devices have $3.0 \mu\text{s}$ to assert DTACK*. However, for devices that implement DMA, it is not this simple. For example, during a DMA output cycle, RAM must be read *and* the I/O card written to within one cycle while BAS* is asserted. Thus, the *combined* RAM read time and I/O card write time must be less than $3.0 \mu\text{s}$ or a bus error will occur. DMA timing is discussed in detail in the DMA chapter.

For a DMA input (I/O read, memory write), DMARDY* must be true (indicating valid I/O card data) within $2.5 \mu\text{s}$ of DMACK* (which goes true at the same time as BAS*). As usual, DTACK* (indicating the memory card has accepted the data) must be true within $3.0 \mu\text{s}$ of BAS* to prevent a BERR* timeout.

For a DMA output (memory read, I/O write), memory is ready within $1.5 \mu\text{s}$ as evidenced by DTACK*; the I/O card has another $1.5 \mu\text{s}$ to accept the data, as evidenced by DMARDY*.

To ensure data transfers without a Bus Error, the following timing requirements must be met.

Description of Time Interval	Maximum Time (in ns)
BAS* low (read or write)	3500
DTACK* low to BAS* high (read or write)	350
Write to Bus Slave:	
BAS* low to DTACK* low (non-DMA write)	3000
DMACK* low to DMARDY* low (DMA write)	3000
Read of Bus Slave:	
BAS* low to DTACK* low (non-DMA source)	3000
BAS* low to DTACK* low (DMA source is memory)	1500
DMACK* low to DMARDY* low (DMA source is I/O)	2500

Interrupt Operation

Even though the MC68000 supports two types of interrupts, only one method of responding to interrupts is currently supported on the DIO Bus: autovectored interrupts. With autovectoring, the interrupting device does not provide a vector to the interrupt service routine -- the Processor Board generates its own default vector. Like the 68000, the DIO Bus supports seven interrupt (hardware) priority levels.

Interrupt Signals

The interrupt signals on the DIO Bus are shown below. Interrupts can occur at levels 1 through 7 (lowest to highest); interrupt levels 3 through 6 are for "external" I/O cards. The following table shows the assignment of interrupt levels for Series 200 computers.

Signal	Hardware Priority (INT LVL)	Device
INT1*	1	Keyboard/Real-Time Clock
INT2*	2	Internal Disc
INT3* - INT6*	3 - 6	External I/O
INT4*	7	Reset Key, Powerfail

Note that there is no INTO* signal: interrupt level 0 is the quiescent (non-interrupting) state. For 68000-based Bus Masters, logic is used to encode these interrupt signals into the three processor inputs, IPL0, IPL1, and IPL2. For example, Series 200 Processor boards use an 74LS148 8-to-3 priority encoder to allow INT1 through INT7 to generate the proper combination of IPL0, IPL1 and IPL2.

The following interrupt levels are the only interrupt levels which have been "hardwired"; all other I/O cards have a two-bit switch to select one of levels 3 through 6.

Internal HP-IB	Level 3
DMA Interface	Level 3
Internal RS-232 (Model 16 only)	Level 4

Note

Even though IR1, IR2, and IR7 are not used for external I/O, the signals have been put on the backplane for expandability and compatibility with future products. *Do not* use IR1, IR2, or IR7 in Bus Slave implementations.

Interrupt Description

The DIO Bus does not support interrupt vectoring from the I/O cards. Instead, autovectoring on the Processor Board is used. The interrupt sequence is described as follows; refer to the subsequent timing diagram as you read about the sequence:

1. When an interrupt request is made on one of the Interrupt Request signal lines (IR1* through IR7*), logic on the Processor board is used to encode the signals into the three inputs IPL0, IPL1, and IPL2 of the 68000 processor. Series 200 processor boards use an 74LS148 8-to-3 priority encoder to allow INT1* through INT7* to generate the proper combination of IPL0, IPL1 and IPL2.
2. If the level of the interrupt is greater than the current processor priority, the processor begins exception processing (after finishing the current instruction).
3. In response to an interrupt request, the Processor Board generates one of 7 vectors that is a function of the interrupt level. These interrupt vectors are mapped in high memory by the Boot ROM. For information about the 7 interrupt vectors, refer to the Boot ROM section in the *Pascal 2.0 System Internals Documentation*, HP part number 09826-90074.
4. If more than one I/O card is on the same interrupt level, then it is not possible to tell which card is interrupting. Thus, a software polling routine must be used to determine which card to service. The two most significant bits of read register 1 contain interrupt information: Bit 7 is set if the card is enabled to interrupt, and bit 6 is set if the card is currently requesting an interrupt.

Utility Signals

This section identifies and defines the signal lines which serve utility-type functions on the DIO Bus. These utility lines supply reset and state-decoding capabilities for the bus.

Signals

The utility signals consist of the following lines:

- RESET*
- HALT*
- Function Codes (BFC0, BFC1, BFC2)

Reset Operations

The RESET* signal goes low at system power-up to allow cards to properly initialize. The Processor board can also generate RESET*.

Power-Up and Power-Down Resets

RESET* is automatically generated within the system at power-up and remains true until 120 ms after +5V reaches 4.5 volts. When RESET* is negated, the +12V supply will have been within its 11.5-volt limit for at least 55 ms. At power-down, RESET* is re-asserted within 15 ms after +5V drops to approximately 4 volts.

Reset by the Processor

The processor can generate a Reset by executing a RESET instruction. See the *MC68000 User's Manual* for further information.

Reset by a Bus Slave

Asserting RESET* by an DIO Bus device does not reset the processor board unless HALT* is simultaneously asserted; thus, RESET* by itself only resets other bus devices.

Note

Bus Slaves should not be allowed to assert RESET* and HALT* signals simultaneously, since such action Resets the entire system.

Function Code Signals

The Function Codes (FC2, FC1, and FC0) generated by the 68000 are buffered (BFC2, BFC1, and BFC0) and brought out on the bus for general purpose use and for future expandability. As would be expected, the Function Code buffer is disabled when bus control is passed; however, this buffer is also disabled during an interrupt acknowledge cycle to inhibit certain control signals which happen to share the same buffer. Therefore, when FC2=FC1=FC0=1 (interrupt acknowledge), the buffered Function Codes on the bus will float (no pull-ups) and are undefined. Therefore, to use Function Codes, the following guidelines should be followed:

- When BAS* occurs, the Buffered Function Codes are valid. BAS* is one of the control signals inhibited by disabling of the buffer during an interrupt acknowledge cycle; so when BFC2 -- BFC0 are floating, BAS* is pulled high with a pull-up resistor.
- During an interrupt acknowledge cycle, the presence of IACK* indicates that FC2=FC1=FC0=1. However, BFC2, BFC1, and BFC0 are undefined and cannot be used.

Note

It is recommended that Bus Slaves not use the Function Codes, even though they are currently defined, since future implementations may not define them.

Electrical Specifications

This section defines the non-timing electrical specifications for the DIO Bus. Included in this chapter are power supply tolerances, card power dissipation specifications, and signal loading information. Pinouts of the DIO Bus are discussed in the "Mechanical Specifications" section.

Power Distribution and Grounding

Power on the DIO Bus is distributed on the backplane as regulated, dc-voltage supplies. The supplies are:

+5 Vdc -- Main logic supply

+12 Vdc -- Provided for I/O circuitry requiring multiple voltages. Can also be used for analog applications.

-12 Vdc -- Provided for I/O circuitry requiring multiple voltages, can also be used for analog applications.

Note

This +12V supply is used for the 9826A's disc drive motor. Designers should be aware of potential noise on the line.

Power Supply Tolerances

The specifications shown below allow for the effects of line regulation, load regulation, cross regulation, initial accuracy, temperature stability, and ripple. The tolerances represent the worst-case tolerances for existing DIO Bus devices.

Supply	Tolerance	Range
+5	+5/-4.3%	5.25, 4.78
+12	+6/-4%	12.7, 11.5
-12	+10/-4%	-13.2, -11.5

Power Requirements of Cards

The following table shows the typical and maximum supply power (and current) available for use with cards used in Series 200 computers and bus expanders.

TYPICAL POWER & CURRENT				MAXIMUM POWER & CURRENT			
TYP. POWER PER CARD (WATTS)	TYPICAL AVERAGE CURRENT (Amps dc)			MAX. POWER PER CARD (WATTS)	MAXIMUM AVERAGE CURRENT (Amps dc)		
	+5	+12	-12		+5	+12	-12
4.4	.8	.096	.064	5.3	.96	.120	.080

Backplane power is a finite resource which makes its availability dependent upon the configuration of the backplane. The diagram above shows standard power requirements for a typical Bus Slave. If a Bus Slave uses more than the standard current requirements, other calculations are needed to ensure power consumption does not exceed power supply design limits. The following table gives the maximum current (and power) allotted to the backplane.

Maximum Typical Amps

	+5	+12	-12
Model 16	2.2 A	0.33 A	0.20 A
	Not to exceed 17.5 Watts total		
Models 26&36	7.6 A	0.91 A	0.60 A
	Not to exceed 41.6 Watts total		

CAUTION

POWER LIMITS DISCUSSED IN THIS SECTION MUST NOT BE EXCEEDED. FAILURE TO FOLLOW THESE GUIDELINES VOIDS ANY WARRANTIES ON THE AFFECTED EQUIPMENT AND DEVICES.

Current requirements for existing interface and accessory cards.

Interface or Accessory	Typical current (mA)		
	+5V	+12V	-12V
HP 98254 64 Kbyte Memory	590	0	0
HP 98256 256 Kbyte Memory	830	0	0
HP 9888A Bus Expander interface	1000	0	0
HP 98620 DMA Controller	1200	0	0
HP 98622 GPIO Interface	750	0	0
HP 98623 BCD Interface	500	0	0
HP 98624 HP-IB Interface	470	0	0
HP 98625 High-speed Disc Interface	600	8	0
HP 98626 RS-232 Serial Interface	400	50	50
HP 98627 Color Video Interface	1100	0	0
HP 98628 Datacomm Interface	720	37	60
HP 98629 Resource Mgmt. Interface	750	37	37
HP 98691 PDI (without EPROM)	750	37	60
HP 98028 Resource Mgmt. Multiplexer (4 channels connected)	530	530	0
HP 13264 Data Link adapter	30	160	23
HP 13265 300-baud Modem	100	45	45
HP 13266 Current loop adapter	200	90	80

Correct voltages are guaranteed from the mainframe power supply only if the above limits are observed (these limits refer to all power supplied to the backplane, including any external devices obtaining power from I/O cards). Even though the system can operate properly for short periods while exceeding these limits, local heating and other stresses that result shorten the life and compromise the reliability of the power supply.

The following examples show power calculations for two configurations (all current is in mA):

Example 1

	+5	+12	-12
1 HP 98620 DMA Controller	1200	0	0
1 HP 98256 256 Kbyte memory card	830	0	0
1 HP 98628 Datacomm Interface	710	37	60
1 HP 13265 300-baud Modem	100	45	45
	2840	82	105

Total power: 16.4 Watts

Conclusion: Total currents are well within the acceptable limits for Models 26 and 36.

Example 2

	+5	+12	-12
1 HP 98620 DMA Controller	1200	0	0
3 HP 98256 256 Kbyte memory car	2490	0	0
4 HP 98628 Datacomm Interface	2840	148	240
4 HP 13265 300-baud Modem	400	180	180
	<hr/>	<hr/>	<hr/>
	6930	328	420

Total power: 43.6 Watts

Conclusion: The value for total power exceeds the maximum. This configuration should not be attempted.

On-Card Fuse Specifications

A UL/CSA/IEC requirement is that any device operating from a supply capable of supplying more than 8 amps be fused. Therefore, any board plugged into the backplane **must** have a 4-amp maximum fuse in series with the +5V bus.

Signal Loading

The following table shows the drive capabilities of each output signal and recommended drivers for each input signal on the DIO Bus.

Signal Name	Maximum Receive Loading	Recommended Send Device
BAS*, BR/W*, BUDS*, BLDS*	1 LS load max.	SN74LS245 buffer
IMA*, DTACK*, ENDT*	1 LS load max.	Any 3S/OC LS gate
DMAR1*, DMAR0*, DMACK1*, DMACK0*, DMARDY*, DONE*	1 LS load max.	Any 3S/OC LS gate
INT6*-INT3*	1 LS load max.	Any 3S/OC LS gate
BFC2*-BFC0*	2 LS loads max.	SN74LS245
RESET*	5 LS loads max.	SN7417 OC Buffer
HALT*	.8 mA TOTAL for card cage	--
BA23-BA1	1 LS load max.	SN74LS245 Buffer
BD15-BD0		SN74LS245 transceiver

Notes

1. "3S" signifies a device with 3-state outputs
2. "OC" signifies an device with open-collector outputs

Mechanical Specifications

This section presents sufficient mechanical information to create printed-circuit (PC) boards that fit into the Series 200 backplane.

Specifications for Cards

The drawing on the following page shows the size requirement for I/O or non-I/O cards. The points worth noting are:

- I/O cards fit in every other slot of the I/O backplane. The slots in between hold non-I/O cards such as RAM cards.
- Non-I/O cards have a recess at the rear to allow clearance for the connector of the next-lower I/O card.
- I/O cards have a "keep out" area in the rear where traces and parts are not allowed to prevent them from shorting to the metal coverplate.
- For both types of cards, space must be left on either side of the board to prevent components from interfering with the card guides in the card cage.
- I/O connectors are left-justified and extended to the right as needed for the size of the connector.

The following drawing shows the outline of an I/O-type board that can be used in the Series 200 card cages.

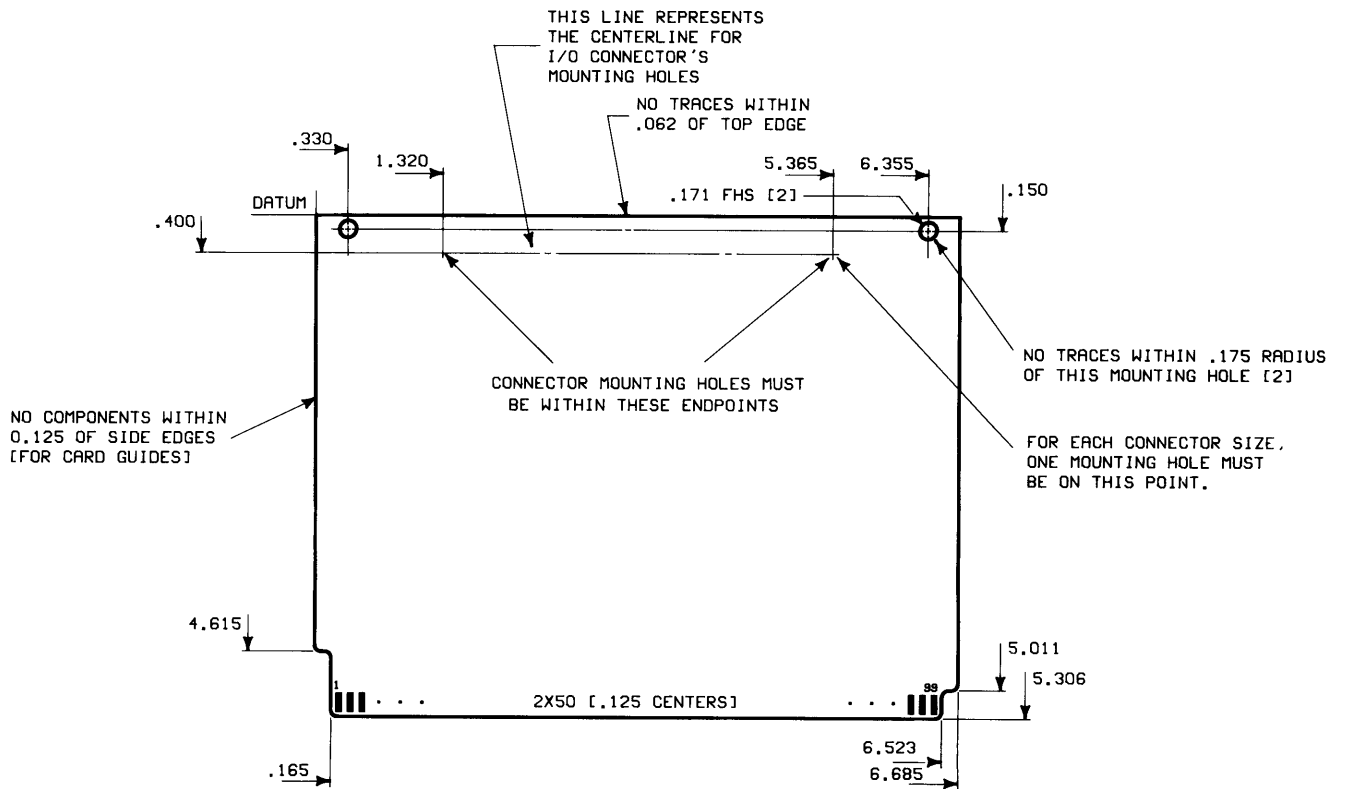


Figure 8. Blank PC Board Outline

Non-I/O-type PC boards must have a cutout to allow room for the connector of the next lower I/O board's connector, as shown in the next drawing.

Card Cage Specifications

Shown below are the specifications of the 9826A/9836A card cage. The drawing also shows the maximum vertical height of components on the boards. The rear panel is further described in the next drawing.

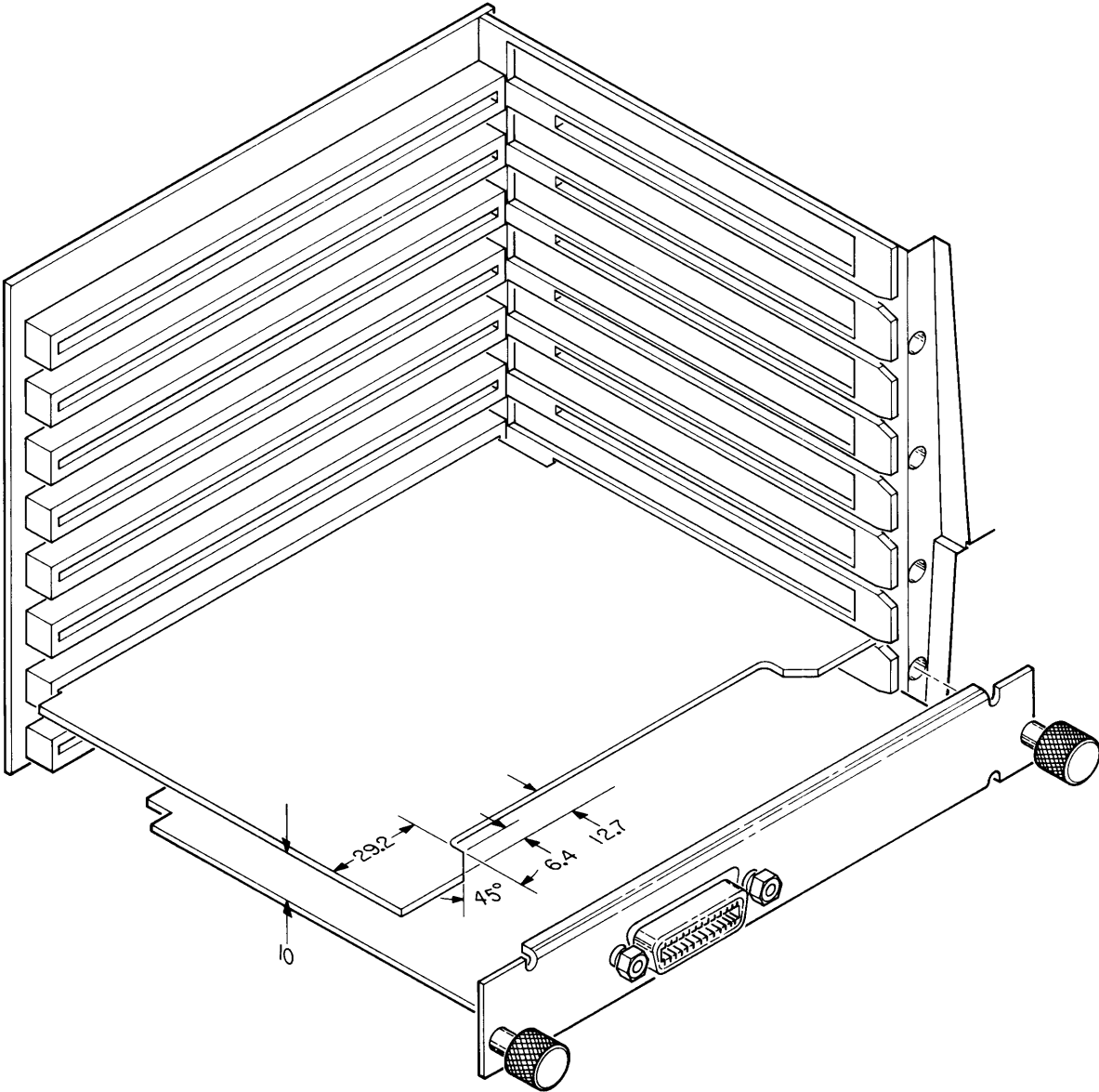


Figure 9. HP Series 200 Card Cage

I/O Card Coverplate

The coverplate for I/O cards is shown in the following drawing. Note the placement of different sizes of connectors.

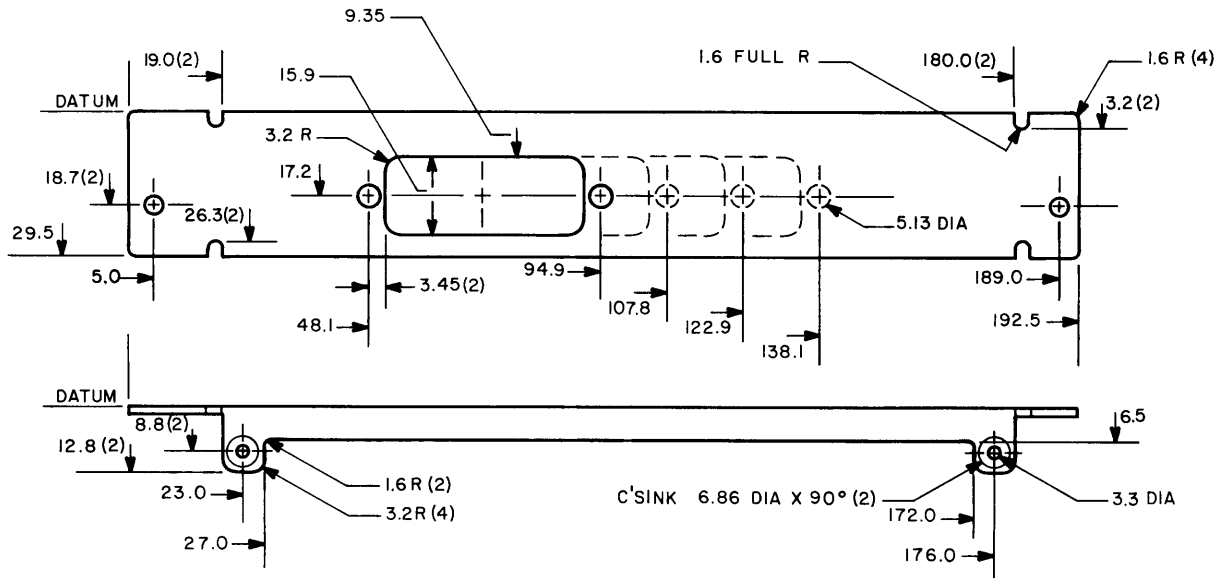


Figure 10. I/O Card Coverplate

Minimizing Electromagnetic Noise

The following rules should help to minimize electromagnetic interference (EMI) problems:

- PC boards should be a minimum of 4 layers, with planes 2 and 3 reserved for power and ground, respectively. Boards greater than 4 layers should maintain power and ground on the middle layers, so that a good, high-frequency bypass capacitor is formed by the planes.
- I/O cards must have a sheet metal coverplate with screws which securely mount the board in the computer and provide good electrical connection to safety ground.
- Grounding of I/O cable housings or shields is done through the coverplate. Appropriate techniques must be used to ensure solid contact between the connector housing and the coverplate. If the shield ground appears on a pin of the connector (as with HP-IB), termination can be done with a generously sized PC trace from the connector to the screwpad which secure the coverplate to the PC board.

PC-Board Layout Rules

The following PC-board layout rules should be met:

- Each card must limit loading to one LS load on the following signals: BA23-BA1, BAS*, BD15-BD0, BUDS*, BLDS*, and BR/W*.
- The PC board trace length for the above signals should be as short as possible and no more than 3 inches. Where possible, these signals should be isolated from the ground and power planes to minimize capacitance.
- PC boards should be a minimum of four layers, with planes 2 and 3 being power and ground, respectively. For boards with more than four layers, the middle layers should be power and ground.
- IC's are mounted parallel to the connector with pin 1 being in the lower left when viewing the board from the component side with connector pin 1 in the lower left corner.
- Adequate bypass capacitors are highly recommended.
- The PC edge connector is a standard S-100 connector with 100 pins on 0.125-in centers and 0.060-in fingers.

Pin Assignments

The DIO Bus pinout is shown in the following table. Odd pins are on the component side, even pins on the circuit side. Relative to viewing the board from the component side with the connector pointing down, the pins numbers increase from left to right.

The following conventions are used in the table:

1. A "-" in front of the pin number indicates should *not* be implemented by a Bus Slave. "Spare" pins should not be used at all.
2. A "#" in front of the pin number indicates an the line is *optional* in a particular subsystem.

Component Side	Circuit Side
1 DMAR0*	2 DMAR1*
3 DMACK0*	4 DMACK*
-5 Spare 0	-6 IR7*
-7 IR2*	-8 IR1*
9 DMARDY*	-10 BG1*
-11 BG2*	-12 BG3*
13 GND	14 GND
15 IR4*	16 IR3*
17 IR6*	18 IR5*
-19 VECTOR*	-20 IACK*
21 GND	22 GND
-23 BG*	-24 BR*
25 #DONE*	-26 BGACK*
-27 Spare 1	-28 Spare 2
-29 BDRV* (BMON*)	30 #ENDT*
-31 BFC0	-32 BFC1
-33 BFC2	34 DTACK*
35 GND	36 GND
37 RESET*	-38 BERR*
39 GND	40 GND
41 #IMA*	-42 FOLD*
43 BLDS*	44 BUDS*
45 BR/W*	46 BAS*
47 GND	48 GND
49 HALT*	50 BA1
51 BA2	52 BA3
53 BA4	54 BA5
55 BA6	56 BA7
57 BA8	58 BA9
59 BA10	60 BA11
61 GND	62 GND
63 BA12	64 BA13
65 BA14	66 BA15
67 BA16	68 BA17
69 BA18	70 BA19
71 BA20	72 BA21
73 BA22	74 BA23
75 GND	76 GND
77 BD0	78 BD1
79 BD2	80 BD3
81 BD4	82 BD5
83 BD6	84 BD7
85 +5 V	86 +5 V
87 BD8	88 BD9
89 BD10	90 BD11
91 BD12	92 BD13
93 BD14	94 BD15
-95 DGND	-96 DGND
-97 Spare 3	-98 Spare 4
99 -12 V	100 +12 V

Operation in the Bus Expander

Devices for the HP 9888 Bus Expander are required to meet the same electrical and mechanical constraints as devices which plug directly into the I/O cardcage. The Bus Expander employs delay lines and latches to ensure that all timing requirements are met. This chapter discusses features of the Bus Expander and several limitations affecting operation of DIO Bus devices in the expander.

Features of the Bus Expander

The features of the Bus Expander are as follows:

- The Bus Expander card plugs into an I/O slot in the computer. Up to 4 Bus Expanders can be plugged into a computer; however, an expander may not be attached to another expander.
- The Bus Expander is totally self-powered.
- The Bus Expander has sixteen slots which will support eight I/O cards and eight non-I/O cards, or sixteen non-I/O cards.
- A 5.2-foot cable connects the Bus Expander to the computer. Signals are buffered at each end of this cable (on the board that plugs into the computer and on the board in the expander). The I'm Addressed signal (IMA*) discussed previously is used to "turn the buffers around" if the addressed card is in the expander.

Operating Limitations With the Expander

In designing DIO Bus devices, designers should be aware of the following limitations when operating in the Bus Expander:

1. Bus Masters cannot operate in the Bus Expander; there is no provision for "turning around" certain signals such as the address bus.
2. RAM boards will operate in the Bus Expander; however, RAM boards require six-state accesses as opposed to five-cycle access for boards installed in the computer. This is due to signal delays. Software being executed from the Bus Expander will thus run proportionately slower and timing loops will be altered.
3. For information only, the 9826/36 Powerfail option (which is installed internal to the mainframe) is not supported with the Bus Expander. The Bus Expander resets the computer when the power fails and again when power returns. This will destroy any data or programs in memory. For correct operation, the expander must be turned on before the computer is powered up and not powered down while the computer is operating.

Design Summary

Many details covering Bus Slave design have been covered in other sections. The key requirements are summarized below.

I/O Card Design Guidelines

The following design guidelines are for external I/O cards only. Following these guidelines is a sample design highlighting key features.

- I/O cards should provide five select code switches to allow select codes settings of 0 through 31.
- I/O cards must implement the four standard I/O registers (and bits thereof) described in the Registers section. Additional registers may be defined as needed.
- I/O cards may be either 8-bit or 16-bit devices.
- Implementing interrupt capability is left to the option of the designer; however, it is strongly recommended that interrupt capability be included. I/O cards that implement interrupt capability must have switches to select interrupt levels 3 to 6.
- Implementing DMA capability is left to the option of the designer. I/O cards that implement DMA can optionally use DONE* from the DMA Controller to determine when DMA is done.

I/O Card Design Example

This section presents an example 8-bit I/O card design and describes the following interface elements: data-transfer, interrupt, and DMA.

Data-Transfer Interface

Figure 11 shows a typical circuit used for transferring data between the I/O device and the Processor. The key steps that occur while accessing the card are as follows:

1. An Address Comparator, the 74LS688, compares the upper 8 address bits (BA23-BA16) to the bit pattern for the External I/O memory space (011) and the user-set select code switches (00000 through 11111). This comparison is enabled by BAS*. The Card Select output (CS*) is generated when the addresses match. CS* does not have glitches, because the address precedes BAS* by 15 ns.
2. CS* generates IMA* and enables the DTACK* buffer. CS* is not used to enable the Data Bus Buffer, the 74LS245, because of a possible driver conflict between the I/O card and the Processor Board. This conflict would otherwise occur during a write cycle when BR/W* is asserted while BAS* is low; BR/W* low would cause the Processor Board buffers to drive the data bus while the I/O card was still driving the data bus (until it recognizes that BR/W* is low, which takes 15-20 ns). To avoid this, the I/O card does not enable its Data Bus Buffer until BLDS* is low.

3. When CS* and BLDS* are both asserted, LDCS* is generated (indicating the start of an access cycle). LDCS* generates DTACK* (after some delay), regardless of whether the operation is a read or a write. The delay time should be set to the longest time required for either reading a register (plus 30 ns setup on the bus) or setting up a register for clocking when DTACK* is asserted. LDCS* also enables the Data Bus Buffer.
4. Reading and writing are then determined by BR/W*, and the steps of both processes are as follows:
 - a. While Reading, BR/W* high enables the Data Bus Buffer to drive the DIO Bus data lines (BD7 through BD0).
 - b. LDCS* low and BR/W* high enables the Register Select chip, the 74LS138, which generates a read strobe for the device at the address selected by BA2 and BA1. The addressed device then drives the I/O card's data bus. As mentioned above, the DTACK* delay circuit should be designed to ensure 30 ns of setup time prior to driving DTACK* low.
 - c. The card remains in this state until BLDS* or BAS* goes high, disabling the Data Bus Buffer and the Register Select chip.
 - a. While Writing, BR/W* causes the Data Bus Buffer to drive the I/O card's data bus.
 - b. LDCS* low and BR/W* low enables the Register Select chip, which generates a write strobe at the device selected by BA2 and BA1.
 - c. Generation of DTACK* ends the write strobe. The data hold time for the register is guaranteed by the minimum response time of the Bus Master to DTACK* low (85 ns).
 - d. The write cycle ends when BLDS* or BAS* goes high.

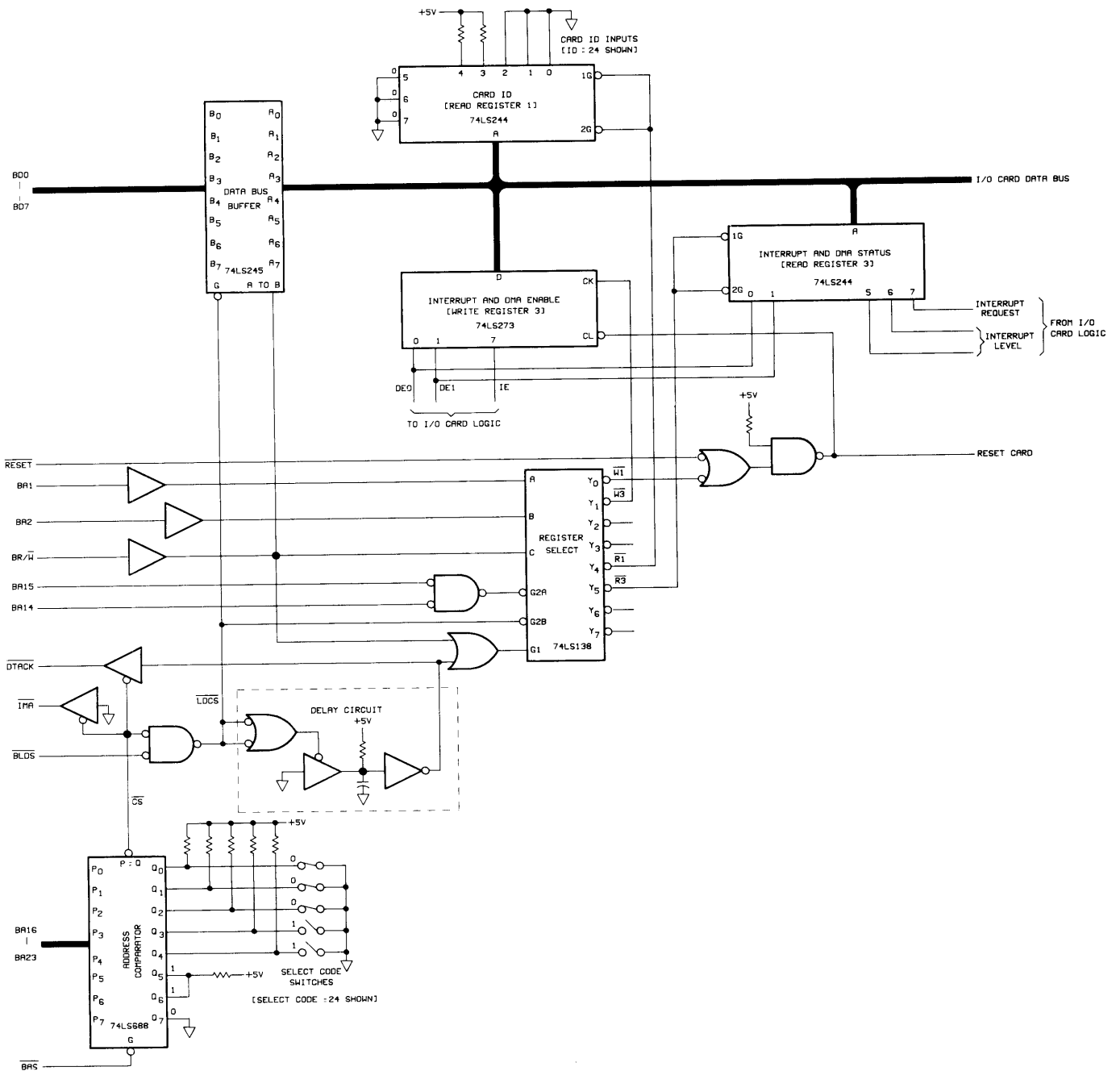


Figure 11. Example Data-Transfer Interface Design

Interrupt Interface

Figure 12 shows a typical interrupt request circuit. The circuit operates as follows:

1. If Interrupt Request (IR) and Interrupt Enable (IE) are true, one of the interrupt request signals (IR3*, IR4*, IR5* or IR6*) will be low, as determined by the two Interrupt Level lines (ILO and IL1).
2. I/O cards are polled by software to determine which card is interrupting. When bit 7 of the Interrupt and DMA Status Register (Read Register 3) is set (to 1), interrupts are enabled; if bit 6 is set (1), an interrupt is being requested by the card (or peripheral connected to the card).

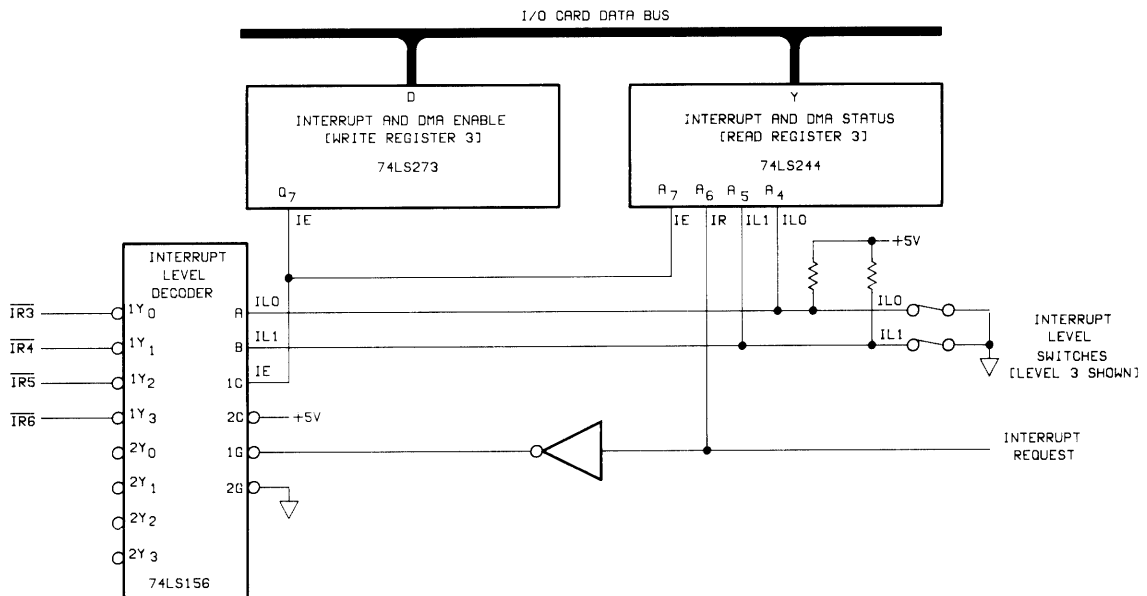


Figure 12. Example Interrupt Interface Design

Note

External interrupt vectoring is not supported with the DIO Bus.

DMA Interface

Figure 13 shows a typical DMA interface. It operates as follows:

1. If a DMA channel is enabled (by setting bit DE0* or DE1* in the Interrupt and DMA Enable Register, Write Register 3), a DMA Request signal from the I/O card drives DMAR0* (or DMAR1*) low.
2. DMACK0* (or DMACK1*) enables the Data Bus Buffer. During a normal R/W operation, the direction of this buffer is determined by BR/W*. During a DMA operation, the exclusive-OR gate inverts BR/W* to control the direction; this is because BR/W* is intended for memory so the I/O card uses it in the opposite way.
3. The I/O card generates DMARDY* just as it normally generates DTACK* during a R/W cycle. For a DMA input cycle, the delay timing starts immediately with BR/W* high. For a DMA output cycle, the delay timing is not started until valid data is on the bus as indicated by DTACK* from the memory device.

Design Qualification

All HP Computers and supporting equipment are designed according to rigorous standards and thoroughly tested to ensure that they meet these high standards. Every new design for an I/O card to be used with Series 200 Computers should likewise be carefully qualified for adherence to acceptable design standards. Qualification for new cards can be broken down into the following areas:

- Safety compliance
- Hardware qualification
- Software qualification

This section outlines qualifications that should be made before finalizing any Bus Slave design.

Safety Compliance

I/O cards must be designed to meet *all* UL, CSA, and IEC safety requirements. This requires that the I/O card's coverplate *always* be secured to the computer chassis with dog bolts and that the I/O connector and cable be adequately grounded to the coverplate. This configuration must meet the following requirements:

1. Ground-current carrying capacity: The conductive path between the I/O cable (cable shield, connector ground pins, and connector shell) and the safety ground pin of the power receptacle must be capable of carrying 30 amperes for 120 seconds. This requirement can be expressed in ohmic resistance for the following two cases:
 - a. For cable lengths less than 4 meters, the dc resistance between the end of the cable and the safety ground pin of the power receptacle should be less than 100 milliohms.
 - b. For cable lengths greater than 4 meters, the dc resistance between the I/O connector on the card's coverplate (cable shield, connector ground pins, and connector shell) and the safety ground pin of the power receptacle should be less than 100 milliohms.

WARNING

I/O CABLES WHICH ARE NOT GROUNDED AS STATED
ABOVE PRESENT A POTENTIAL SHOCK HAZARD TO
THE USER OF THE EQUIPMENT.

2. Fault-current carrying capacity: Because the +5V supply in the Series 200 computers has over an 8-ampere fault-current capability, an on-card fuse is required. Refer to the "Electrical Specifications" section for the recommended fuse.

CAUTION

AN I/O CARD NOT EQUIPPED WITH THE PROPER FUSE
IS CONSIDERED TO BE A MISUSE OF THE EQUIPMENT
AND MAY RESULT IN A PERSONAL HAZARD TO THE
OPERATOR AND/OR EQUIPMENT DAMAGE.

Hardware Qualification

Hardware testing can be divided into two areas: environmental testing and configuration testing. Environmental testing involves testing the I/O card throughout the range of operating environments. Configuration testing involves testing several different I/O operations with all significant mainframe configurations.

In performing environmental tests, a subset of possible configurations are selected for testing in the following areas:

- Initial testing of a small sample of prototypes: Testing at this stage involves high electrical, thermal, and mechanical stresses of short duration, often to the level of inducing failures so as to identify weak points of the hardware.
- Strife (stress+life) testing of a larger sample of production devices: Testing at this stage involves thermal cycling and vibration tests derived primarily from MIL-Std-810B. These tests also involve forcing failures, determining the cause, and implementing a solution. However, the emphasis is on determining failure modes and assessing margins. It is also the first opportunity to search for production-process related failure mechanisms.

Approximately 50 thermal cycles are performed with the range of temperatures varying from -20 to 65 °C. The time for each cycle is adjusted so that the units reach thermal equilibrium during the dwell portion of each cycle.

Random vibration tests are performed periodically with a range of accelerations from 1-2 g (9.8-19.6 m/s/s) rms. Total accumulated time is on the order of 25 minutes.

- Environmental testing of a small sample of units: This stage of testing involves a complete set of tests known as HP Class B (Industrial and Commercial) Environmental tests. Testing is performed in the following areas for adherence to the stated standards:

Type of Test	Description
Temperature:	
Non-operating (storage)	-40 to +75 degrees Celsius
Operating (survival)	-20 to +65 degrees Celsius
Normal Operating	0 to +55 degrees Celsius
Humidity:	
Operating	40 degrees Celsius at 5 to 95% Relative Humidity
Non-operating	65 degrees Celsius at 90% Relative Humidity
Condensation	Operates without damage and recovers within specified limits.
Vibration:	
Cycle Range	5-55-5 Hz.
Amplitude (p-p)	0.38 mm
Sweeptime	15 min. (1 min./octave)
Dwell @ Resonances	10 min. at each resonance
Ampl. @ Resonances	3.17 mm @ 5-10 Hz. 1.52 mm @ 10-25 Hz. 0.38 mm @ 25-55 Hz.
Shock:	
Magnitude	30 g (approx. 294 m/s/s)
Duration	11 ms
No. Shocks	18 (3 on ea. of 6 surfaces)
Waveform	Half-sine
Bench handling	102 mm tilt drop
Altitude:	
Non-operating	15 300 m
Operating	4 600 m

- RFI testing: This type of testing requires compliance with VDE Level B (with a 2 db margin) and FCC Class B standards.

Software Qualification

The key concern with software is, of course, its reliability. Sufficient testing should be performed to ensure that the card operates properly with the desired operating system(s). When operating systems are revised, new operating systems are released, and when changes are made to an I/O card which affect its operation, additional software testing should be performed.

Subject Index

a

Access methods 110,115
Address record 373
Alternate DAMs 15
APPEND 71
Architecture 444
Auto configuration 13

b

BASIC Files 19
BATTERY 402
Battery backup 361
Beeper 295,309
BOOT 423
Boot command 438
Boot device 12
Boot disc format 387
Boot files 88
Boot request 422
Boot ROM 381,406
Boot ROM calls 435
Boot:
 File names 88
 Linking 85
 Loading 85
 Memory map 89
 Overview 87
 Summary 100
Buffer I/O 452,465
Bus:
 Addresses 12
 Error 551
 Expander 568,568
 I/O 523
 Pinout 567
 Signals 534
 Slave 525
 Timeout 551

c

Card cage 563
Clock 296
CLOSE 72
Command interpreter 275
Configuration 14,400
CPU load state 408
CRASH 429
CRT:
 9816 328
 9826 327
 9836 326
 Controller 330
 Cursor control 334,335
CRTCLEAR 426
CRTID 403
CRTINIT 425
CTABLE 5,31

d

DAMs:
 Alternate 15
 Definition 12
 LIF 231
 Primary 16
 Reference 221
 WS1.0 17
Data Comm drivers 463
DEF 369
Default mass storage 394
Define source 369,378
Definition symbol table 377
Device specifier 12
DIO bus 523
Direct access files 75
Directory access methods 12,114,221
Directory entries 17
Directory path names 63

Disc:	
Drivers	415
Drives	18,345
Errors	416
Format	391,392
Partitioning	43
Display:	
Alpha	328
Attributes	329
Drivers	329,333
Graphics	328
Pascal access	334
Processor registers	330
Remote	466
Toggling the screen	337
DMA	544
DMA drivers	462
Drivers:	
Addition	494
Modification	498
Removal	493
Structure	454
DTACK	534
Dump alpha	337
Dump graphics	337

e

Errors:	
File IO	142
Exception vectors	440
Export text	369
External symbol table	376

f

FIBs	108,118
File support	133
File system	107
File:	
Buffer	74
Names	61,64,65,66
Operations	67
Parameters	80
Pointer	69,74
Specifications	61
States	69

Files:	
Access rights	83
Concurrent access	81
Creating	70
Debugging	84
Direct access operations	75
Disposing	72
Example calls	248
Operations	109
Sequential operations	73
Support level	112
System calls	144,247
TEXT	76
Variables	108

g

General value record	373
Global variable	370
GPIO drivers	462
GPIO interrupts	509
GRAPHICS file	22
GVR	373

h

HP-IB:	
Drivers	461
Interrupts	503

i

INITLIB	20,24,53
Interface	11
INTERFACE file	21
Interface text	369
Interrupt	553
Interrupt service routine	282
Interrupts	281
I/O Bus	523
I/O card qualification	574
IO file	22
I/O map	442
ISR	282
ISR procedures	502

k

KBD	323
KBDHOOK	299
Kernel	101,444
Keyboard remapping	297
Keyboard:	
Capabilities	291
Command processing	318
Commands	311
Electronics	307
Example	303
Interrupts	310
Large	300
Module KBD	323
Scanning	307
Service request	320
Small	301
Knob	293,309

l

Librarian	30
LIBRARY	20,23
Library directory	370
LIF	5,388
LIF File names	234
LIF Module	233
LIFDAM	231
Linking	85
Load state	408
Loaders	409

m

Machine configuration	400
Map:	
High ram	430
Low ROM	433
Mass storage	18,394
Memory map	89,528
MISCINFO file	339
MODCAL	8
Module directory	371,371
Modules	27,101,369
Modules in the Kernel	103

n

NDRIVES	406
NMI	428

o

Object code format	369
OPEN	71

p

Partitioning	43
Pascal 1.0	4
Path names	63
Peripheral	11
Power requirements	558
Power-fail option	361
Power-up	458
PROM	398

r

RAM map	430
READ	73
Read cycle	535
REF	369
REF table	378
Registers	531
Remote console	466
Reset	555
REWRITE	70
ROM headers	383
ROM map	433

S

Screen height	335
Screen width	335
SDF	391
Select code	11,442
Select code addresses	530
Sequential files	73
Serial interrupts	514
SID	1
Software tools	8
SRM:	
Access rights	83
Concurrent access	81
Overview	45
Stop key	459
Structured Disc Format	391
Supported mass storage	18
Symbols	370
SYSFLAG	401
SYSFLAG2	402
System volume	12
System:	
Creation	437
Names	6
Switching	422

t

TEXT files	76
TEXT record	378
Third parameter for files	80
Transfer	534
Transfer methods	113
Transfer procedures	499
Trap vectors	440
Typeahead buffer	336

u

Unit table	12,14,113,126
------------------	---------------

v

Variable allocation	9
Volume	12
Volume names	62

w

WRITE	73
Write cycle	538

