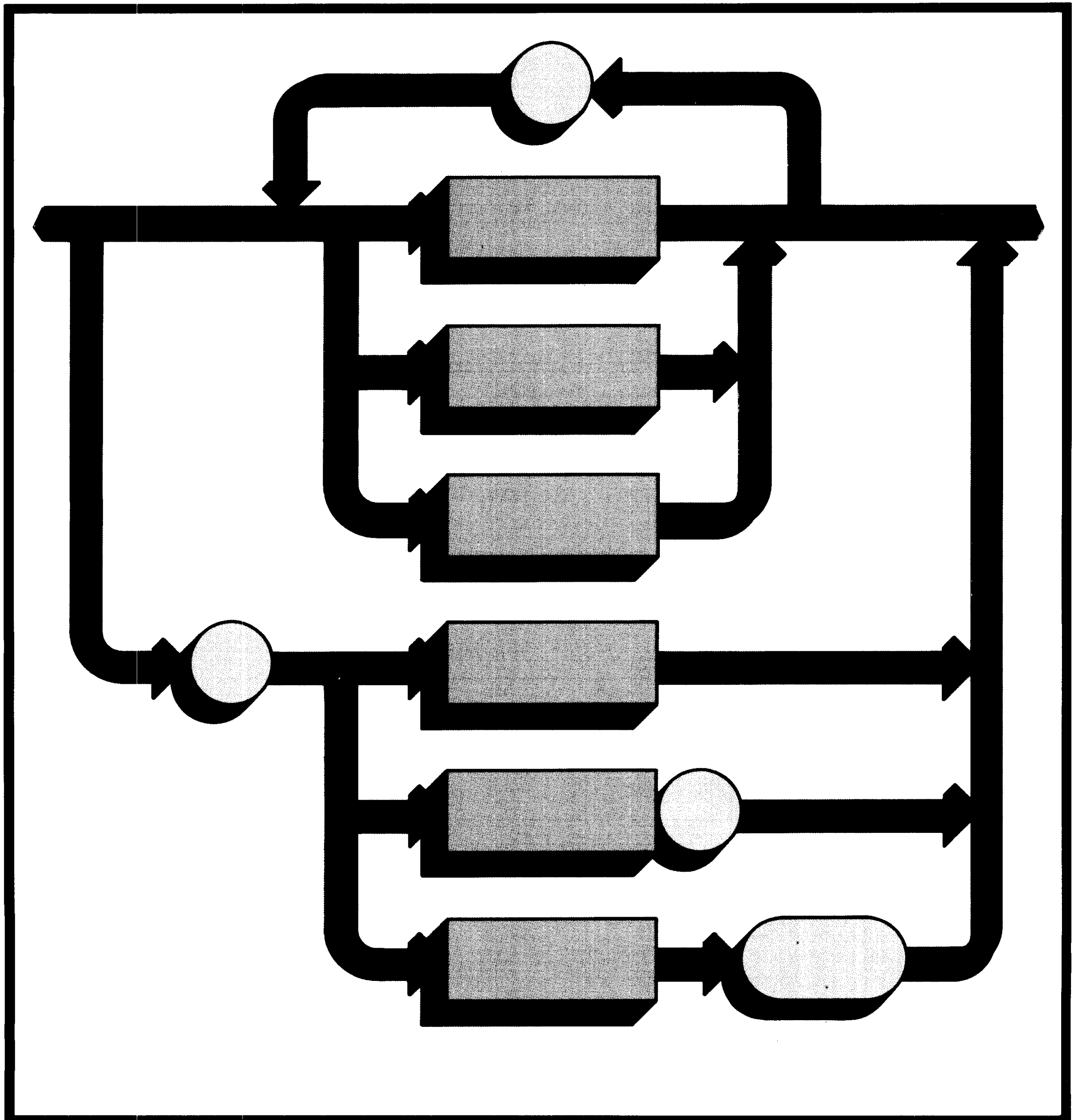
HP 9000 Series 200 Computers

HEWLETT PACKARD

Pascal
System Designer's Guide



Pascal 3.0
System Designer's Guide
for the HP 9000 Series 200 Computers

Manual Part No. 98615-90074

© Copyright 1985, Hewlett-Packard Company.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs 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).

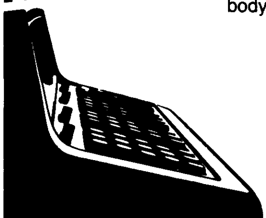
© Copyright 1980, Bell Telephone Laboratories, Inc.

© Copyright 1979, 1980, The Regents of the University of California.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California.

© Copyright 1979, The Regents of the University of Colorado, a body corporate.

This document has been reproduced and modified with the permission of the Regents of the University of Colorado, a body corporate.



Hewlett-Packard Company
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.

February 1985...Edition 1

Warranty Statement

Hewlett-Packard products are warranted against defects in materials and workmanship. For Hewlett-Packard computer system products sold in the U.S.A. and Canada, this warranty applies for ninety (90) days from the date of shipment.* Hewlett-Packard will, at its option, repair or replace equipment which proves to be defective during the warranty period. This warranty includes labor, parts, and surface travel costs, if any. Equipment returned to Hewlett-Packard for repair must be shipped freight prepaid. Repairs necessitated by misuse of the equipment, or by hardware, software, or interfacing not provided by Hewlett-Packard are not covered by this warranty.

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.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR CONSEQUENTIAL DAMAGES.

HP 9000 Series 200

For the HP 9000 Series 200 family, the following special requirements apply. The Model 216 computer comes with a 90-day, Return-to-HP warranty during which time HP will repair your Model 216, however, the computer must be shipped to an HP Repair Center.

All other Series 200 computers come with a 90-Day On-Site warranty during which time HP will travel to your site and repair any defects. The following minimum configuration of equipment is necessary to run the appropriate HP diagnostic programs: 1) ½ Mbyte RAM; 2) HP-compatible 3½" or 5¼" disc drive for loading system functional tests, or a system install device for HP-UX installations; 3) system console consisting of a keyboard and video display to allow interaction with the CPU and to report the results of the diagnostics.

To order or to obtain additional information on HP support services and service contracts, call the HP Support Services Telemarketing Center at (800) 835-4747 or your local HP Sales and Support office.

* For other countries, contact your local Sales and Support Office to determine warranty terms.

Table of Contents

Chapter 1: Introduction

System Internals Documentation 1
Fair Warning 2
How to Use This Documentation 4
  Prerequisites 5
Differences Among Pascal Releases 6
  System Distribution 7
  The CTABLE Program 7
  File System 7
  Object Code Incompatibility 8
  New Peripheral Support 9
  Miscellaneous 9
Software Tools Used for System Generation 10
  Assembler and Librarian 10
  Pascal Compiler 10
  Memory Allocation of Variables 11

Chapter 2: The Booting Process

Introduction 13
  Concepts of Linking and Loading 13
Overview of the Booting Process 15
  How the Boot Files are Chosen 17
Memory Map Development 18
  Summary of the Booting Process 28
The Pascal Kernel 29
  Refresher on Pascal Modules 29
  Modules in the Kernel 32
  Digression on a Trick 34

Chapter 3: The File System

Introduction 35
  Representation of File Variables 36
  High-Level File Operations 37
  The Access Methods 38
  The Unit Table 40
  The Transfer Methods 40
  The Directory Access Methods 41
  How the Access Method is Selected 42
Fields of a FIB 45
  Variant Fields 51
The Unit Table 52
  The Fields of a Unit Entry 53

Chapter 4: File Support

Introduction 59
Error Reporting by the File I/O Subsystems 69
File System Exports 72
doprefix 75
fanonfile 76
fblockio 78
fbufferref 80
fclose 81
fcloseit 82
feof 83
feoln 84
fget 85
fgetxy 86
fgotoxy 87
fhpopen 88
fhpreset 91
findvolume 93
finitb 94
fixname 95
fmaketype 96
fmaxpos 98
foverfile 99
foverprint 101
fpage 102
fposition 103
fput 104
fread 105
freadbool 106
freadbytes 107
freadchar 108
freadenum 109
freadint 110
freadln 111
freadpaoc 112
freadreal 113
freadstr 114
freadstrbool 115
freadstrchar 116
freadstrenum 117
freadstrint 118
freadstrpaoc 119
freadstrreal 120
freadstrstr 121
freadstrword 122
freadword 123
fseek 124
fstripname 125
fwrite 126
fwritebool 127

fwritebytes 128
fwritechar 129
fwriteenum 130
fwriteint 131
fwriteln 132
fwritepaoc 133
fwritereal 134
fwritestr 135
fwritestrbool 136
fwritestrchar 137
fwritestrenum 138
fwritestrint 139
fwritestrpaoc 140
fwritestrreal 141
fwritestrstr 142
fwritestrword 143
fwriteword 144
scantitle 145
suffix 147
zapspaces 148

Chapter 5: Directory Access Methods
Reference Specification for Directory Access Methods (DAMs) 149
The Golden Rule for DAMs 150
Calling DAMs 150
The LIF Directory Access Method 158
Implementation of LIFMODULE 160
LIF Directory File Names 161
Routines within Procedure LIFDAM 162
Details on Various DAM Requests 167

Chapter 6: File Operations
Introduction 173
Filepack Examples 174
Function MIN 175
Function IOERRMSG 175
Procedure IOCHECK 175
Procedure BADIO 176
Function UNITNUMBER 176
Function SAMEDEVICE 176
Procedures ANYTOMEM and MEMTOANY 177
Procedure SETUPFIBFORFILE 180
Procedure CLOSEINFILE 182
Procedure CLOSEOUTFILE 183
Procedure FILECOPY 183
Procedure VOLUMES 187
Procedure REPACK 188
Procedure OPENDIR 188
Procedure CLOSEDIR 189
Procedure CREATEDIR 189

Procedure MAKEDIR ... 190
Procedure MAKEFILE ... 191
Procedure ENDCAT ... 191
Procedure STARTCAT ... 192
Procedure CAT ... 194
Procedure DUPLICATE ... 195
Procedure REMOVE ... 196
Procedure CHANGE ... 196
Procedure ENDPASS ... 197
Procedure STARTLISTPASS ... 197
Procedure LISTATTRIBUTE ... 198
Procedure LISTPASSWORD ... 199
Procedure CHANGEPASSWORD ... 200
Procedure PREFIX ... 201

Chapter 7: CPU Interrupt Handling
Introduction ... 203
Hooking in Your Own ISR ... 208
A Cautionary Note ... 209
Restrictions on Interrupt Service Routines ... 210
Error Conditions "Thrown Away" ... 210
The "ISR in an ISR" Mistake ... 210

Chapter 8: The Keyboard
Introduction ... 211
Summary of Keyboard Controller Capabilities ... 212
Keyboard "Cooking" and "Raw" Data ... 214
Keyboard Access with the File System ... 215
Echoing Read ... 215
Non-Echoing Read ... 216
The Beeper ... 217
Easy-to-Use Extensions ... 217
Avoiding "Hanging Reads" ... 217
Timing with the System Clock ... 218
Using the System Clock and Calender ... 218
Remapping the Keyboard ... 219
Here is What You Want to Know ... 222
Gritty Details of the Keyboard ... 229
About the Electronics ... 229
Keyboard Microcomputer ... 229
Clock ... 229
Circuits Common to the 98203 Keyboards ... 229
Protocol for Keyboard Handling ... 230
Communication Addresses ... 230
Interrupting the 68xxx ... 230
Sending a Command to the 804x ... 231
Processing an 804x Service Request ... 231
Using the Four-Voice Sound Generator ... 233
"Black Box" Description of Functions ... 237
Generalities ... 237

Load Timer Output Buffer Commands 238
Data Request Commands 239
Set-Up Commands 242
Trigger Commands 245
Keyboard Command Processing 246
Knob and Timer Details (8041) 251
The Keyboard at Power-up and Reset 251
Pascal Interface to the Keyboard 252
804x Code Revision and Features Identification 252
The Interface to HP-HIL Devices 253
The Interface 254
The New Keymap 261

Chapter 9: Displays
Introduction 265
"Alpha" Displays 266
Display Hardware Capabilities 267
Alpha Screen Driver Considerations 275
"Bit-Mapped" Displays 276
Display Hardware Capabilities 277
Alpha Screen Driver Considerations 278
Alpha Displays 278
Controlling the Model 237 Bit-Mapped Display 282
The Replacement Rule 282
Using the Line-Mover 283
Model 237 Frame Buffer Allocation 285
Caveats 286
Graphics Screen Driver Considerations 287
Pascal Access to the CRT 289
File System Operations 289
Scrolling 290
Lower-Level Access to the CRT 290
Cursor Motion 290
Interrogating the Dimensions of the CRT 291
Turning the Screens On and Off 291
Dumping the Alpha or Graphic Screens 291

Chapter 10: Internal Disc Drives
Floppy Control Board 293
Theory of Operation 293
Status and Control Registers 296
On-Board RAM (256-byte buffer) 299
Commands and Status 300
Type I Command Flags 300
Type II Command Flags 301
Type III Command Flags 302
Type IV Command Flags 302
Type I Commands 303
Type II Commands 304
Type III Commands 305

Type IV Commands ... 306
Status Information ... 306
Programming Considerations ... 308

Chapter 11: The Boot ROMs
Introduction ... 309
Overview ... 310
Boot Formats ... 312
ROM Headers ... 312
Boot Disc Formats ... 316
LIF System File Format ... 318
SDF Boot Area Format ... 322
UNIX Boot Area Format ... 323
ROM/EPROM Pseudo-Disc Format ... 324
SRM System Files ... 324
Default Mass Storage ... 325
CPU Board ID PROM ... 329
Machine Configuration ... 332
SYSFLAG ... 332
SYSFLAG2 ... 333
BATTERY ... 333
Device Configuration Identification ... 334
CRTID, CRT Presence, Graphics Presence ... 335
Keyboard ... 338
NDRIVES ... 338
Boot ROM Configuration and Revision ... 339
Power-Up Options ... 340
Memory Test Length ... 340
Self-Test Looping ... 340
50/60 Hz CRT ... 341
Configure Mode Software Override ... 341
CPU State at Load ... 343
Read Interface and Secondary Loading ... 344
Flexible Disc Drivers ... 350
System Switching ... 357
CRTINIT ... 359
NMI_DECODE ... 360
CRASH ... 361
Character Table ... 361
High RAM Map ... 362
Low ROM Map Exception Vectors ... 366
Using Boot ROM Routines from Pascal ... 369
Creating a Bootable System ... 370
Guidelines for System Creation ... 371
Rules for Using the Boot Command ... 372
An Example ... 372
Trap/Exception Vectors used in Pascal ... 374

Chapter 12: DGL Internals

Introduction 375
The History and Philosophy of DGL 375
The Structure of DGL 376
IMPORT Hierarchy 378
DGL Modules 381
Changes From Previous Implementations 381
DGL Responsibilities 382
DGL's Graphics Control Block 382
Other DGL Variables 383
DGL Initialization 383
DGL Errors 384
Locator Echoes 385
Specific DGL Tasks 386
GLE Responsibilities 388
What Is GLE? 388
GLE's Graphics Control Blocks 389
GLE Modules 391
GLE Initialization 392
Features of GLE 396
Example GLE Program 397
Drivers 404
Functional Description 404
Syntactical Description 405
Driver Data Structures 406
HPGL Move Example 407
Graphics System Initialization 409
GRAPHICS_INIT 409
DISPLAY_INIT 410
Raster Display Initialization 413

Chapter 13: Floating Point Card

Introduction 417
National's Hardware 418
The Pascal Workstation's Design and Interface 420
Talking to the Card 423
Writing Data to the Card 423
Removing Data from the Card 424
Performing Operations on the Card 425
Waiting 426
Interrupts 427
Memory Map 428
Creating Pseudo-Instructions 429
Long Reals (Longs) 429
Short Reals (Floats) 430
Register-Register Moving 430
Register-Memory Moving 431
Status and Control 434
Programming Examples 436
Powerup/Reset 436

Error Handling 436
Bogus Reads 436
Moving Data Into the FPU 437
Moving Data Out of the FPU 439
Saving and Restoring Context 440
On-Board FPU Operations 441
Putting it All Together 441
Operating System Modifications 442
Differences 442
Debugging 443
Floating Point Instruction Recognition 443
Floating Point Instruction Knowledge 443
Reading and Altering The Floating Point Registers 444
Pseudo-Instruction Table 445
Long-Real Operations 445
Short-Real Operations 446
Conversion-Moves Between FPU Registers 450
Non-Conversion Moves Between FPU Registers 450
Moves Between 68xxx and FPU 451

Chapter 14: Object Code Format
Introduction 453
Purposes of the Object Code Format 453
Definitions 453
Structure of a Library File 454
Library Directory 454
Module Directory 455
General Value or Address Record (GVR) 457
Flags 458
Reference Pointer 459
How a GVR is evaluated 460
EXT Table (External Symbol Table) 460
DEF Table (Definition Symbol Table) 461
Define Source 461
TEXT Record 462
REF Tables 462
Miscellaneous Notes 463

Chapter 15: Device I/O
Introduction 465
The Hardware View 466
The Programmer View 467
General Architecture 468
Main Data Structures 471
ISC_TABLE 471
Driver Read/Write 474
Buffer Control Block 476
Driver Structure 478
High-Level Routines 481
Execution Walkthrough 482

Power-Up ... 482
Stop Key ... 483
Program Compilation and Execution ... 483
Low-Level Drivers ... 485
HP-IB ... 485
GPIO ... 486
DMA ... 486
Data Comm (98628A/98629A) ... 488
I/O Examples ... 490
Using Special Buffers ... 490
Remote Console Driver ... 491
REMKEYS.TEXT ... 497
REMCRT.TEXT ... 501
Removal of Drivers ... 508
Addition of a Driver ... 509
A Specific Example ... 509
Modification of a Driver ... 512
End-of-Transfer Procedures ... 513
Interrupt Service Routine Procedures ... 515
HP-IB Interrupts ... 516
GPIO Interrupts ... 522
Serial Interrupts ... 527

Introduction 1

System Internals Documentation

You are reading the *Pascal 3.0 System Designers' Guide* to the HP 9000 Series 200 Computers. It is one part of the *System Internals Documentation* (SID). The Internals Documentation includes:

- *System Designers' Guide*
- *Assembly Language Source Code Listings (Volume I)*
- *Pascal Source Code Listings (Volume II)*
- *Utilities Disc*
- *Accessory Development Guide*

The *System Designers' Guide* is a collection of engineering notes and reference specifications describing the software interface to the hardware systems supported by Pascal 3.0. Inside you will find detailed information covering a broad range of topics—to quote one of the designers, "enough information to make you dangerous." Reading the following warning may protect you from costly mistakes.

The listings provide you with examples of the actual source code of the Pascal 3.0 Language System.

Fair Warning

If you use the information in this document, you are writing hardware-dependent 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 thereof, 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! Recompilation is always necessary, however, if the "major revision" number—the digit to the left of the decimal point—changes.) 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 some time 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.

The Right Approach for Writing Assembly Language Routines

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 you need to. Don't worry about speed; most people are amazed by the computational performance of the 68xxx processors.
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 I/O Library, you're driving a luxury car. It is very easy to drive, but it wallows around the curves. You might process the voltmeter readings as fixed-point numbers (scaled integers) instead of floating-point numbers, and do character I/O directly by calling a lower level of the I/O Library, thus avoiding some overhead.

   Going to a lower-level entry into the system is an option which doesn't exist with most interpreted language systems, such as BASIC. 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 3.0 Workstation System* 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 it 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. Both the listings volumes also include cross-references to Pascal or assembly language identifiers.

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; but suppress other parts. What is presented in detail is the following: I/O 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; DGL; floating-point math card (HP 98635A) support.

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" (DAMs) and "Access Methods" (AMs). 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 some material on the hardware[1]. A typical reason for using the information published here might be to write and install a device driver for a non-HP 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 that information.

[1] For more in-depth information on the hardware, see the *Series 200 Accessory Development Guide*, also for the Series 200.

NOTE

In this set of manuals, the term **68xxx**, in reference to the CPU of the Series 200 machine, means either the MC68000 or the MC68010, from Motorola.

Also in this set of manuals, the term **804x**, in reference to the keyboard processor of the Series 200 machine, means either the 8041 or the 8042, made by Intel.

Prerequisites

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 3.0 Workstation System* manual (concentrate on modules and system programming language extensions);
- *Pascal 3.0 Procedure Library* manual (know the concepts of physical device I/O); and
- *MC68000 User's Manual* (know your computer's CPU).
- Many examples will also require you to understand 68xxx Assembly language.

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

Differences Among Pascal Releases

The material in this manual describes the internal organization and specifications of release 3.0 of the Pascal language system for HP Series 200 desktop computers.

Pascal 3.0 is very similar to Pascal 2.x in most respects, the most notable differences being in the human interface to the internal peripherals: the keyboard controller and CRT. Of course, Pascal 3.0 supports more external peripherals than does Pascal 2.x.

Pascals 2.x and 3.0 are substantially the same; there are very substantial differences between the internal structures of the 1.0 and 2.x releases. In either case, this material is not a good guide to the innards of previous implementations; to use it that way would be very misleading. Don't try.

The main differences between the 1.0 and 2.x releases are:

- Organization of discs on which the system is distributed,
- Peripheral configuration (`CTABLE` Program),
- File system,
- Object code format,
- I/O Drivers,
- New peripheral support, and
- Miscellaneous.

The main differences between the 2.x and 3.0 releases are:

- Organization of discs on which the system is distributed,
- Built-in peripheral access,
- Peripheral configuration (`CTABLE` Program),
- Some I/O Drivers,
- New peripheral support, and
- Miscellaneous.

Each of these issues is discussed in the following sections, with the exception of `CTABLE`, which is covered in the *Pascal 3.0 Workstation System* manual.

System Distribution

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

Pascal 2.0 was distributed on six discs. The system library file is almost empty, and the I/O, 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 I/O 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.

Pascal 2.1 was also distributed on six discs. The system library file is almost empty, and the I/O, 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 less commonly needed drivers are supplied on a separate disc.

Pascal 3.0 is distributed on ten discs. Again, the system library file is almost empty, and the I/O, Graphics and OS interface modules are supplied on separate discs. Pascal 3.0 supports more peripheral devices than Pascal 2.0 or 2.1. The less commonly needed drivers are supplied on a separate disc.

Consequently, before compiling or running programs which do device I/O 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 3.0 scans interfaces for various peripherals and automatically configures itself, similar to Pascal 2.x. Auto-configuration is discussed in the *Pascal 3.0 Workstation System* manual.

File System

The Pascal 3.0 file system is very similar to the Pascal 2.x file system. For discussion on the file system, see the *Pascal 3.0 Workstation System* manual.

Object Code Incompatibility

Object code is incompatible—non-executable—between "major revisions." That is, a program or module compiled on one major revision will not run on another major revision; it is *object code incompatible.* However, it still may be *source code compatible*, which means that a simple recompilation would suffice to port the program to the other major revision of the operating system.

A "major revision" is a turn of the operating system such that the number to the left of the decimal point changes. For example, the turn from Pascal 2.0 to Pascal 2.1 is *not* a major revision, and thus, the object code is compatible between these two (non-major) revisions. However, the turn from Pascal 2.1 to Pascal 3.0 *is* a major revision, and thus programs and modules will at least have to be recompiled.

The loader enforces the rule about running code only from the same major revision of the operating system. If you attempt to execute an object file which was compiled on another major revision of the operating system, you will get the following message:

```
incorrect version number
```

There are several reasons (not all of which are listed here) which warrant a major revision:

- Changes in global constants, types, or variables; for example, `CRTIREC`.
- Entry points being added, deleted, or their functionality changed. For example, the Pascal 2.x procedure `KBDCOMMAND` has been replaced by `SENDCMD`, `SENDDATA`, and `CMD_READ_1` in Pascal 3.0.
- Other miscellaneous side effects. For example, cache memory in Pascal 3.0; it did not exist in previous versions.

The 2.x and later Filer and Librarian can deal with code files from any major revision of the operating system, and the 2.x and later Filer, Assembler, Compiler, and Editor can deal with source files from any version although Pascal 3.0 introduced some new compiler directives.

New Peripheral Support

The following peripherals are supported by Pascal 3.0.

- The CS-80 discs (7908 family) are supported, including the streaming backup tape drive.
- Subset 80 mass storage devices: 913XD, 9122, etc.
- The Shared Resource Manager is fully supported.
- The 8920x and 9121 flexible disc drives are supported.
- Several new versions of the 913x 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 3.0 system can be 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 68xxx's "supervisor mode". Since the 2.0 release, user programs now run in "user mode," using the USP (User Stack Pointer). Interrupts run in "supervisor mode," using the SSP (System Stack Pointer). 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. Many Boot ROM calls require cache to be off.

SYSDEVS

One major difference between Pascal 2.x and Pascal 3.0 is the software interface to the internal peripherals. The new module `SYSDEVS` contains export text and hooks for entry points for what used to be 5 modules in Pascal 2.x (`KBD`, `KEYS`, `CRT`, `BAT`, and `CLOCK`). This is not a pure move of export text, etc.; many additions, changes and deletions were made in the process. Module `KBD` no longer exists in Pascal 3.0, and `CRTB` has been added. Also, none of the driver modules has export text or even DEFs. They "hook themselves into the system" by pointing `SYSDEVS` procedure variables at their entry points, and are executed solely through hooks. Data structures that used to be "owned" by the modules are now exported from `SYSDEVS` in many cases.

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 3.0.

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 `$MODCAL$`, standing for "modified Pascal." "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 Compiler chapter of the *Pascal 3.0 Workstation System* manual.

The remaining Modcal features (not enabled by `$SYSPROG$`) are used only where necessary[2] in the Pascal 3.0 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 3.0 Workstation System* 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 Assembler chapter in the *Pascal 3.0 Workstation System* manual has examples.

[2] And we found it necessary well nigh everywhere. . .

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 68xxx 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, so you can deal with I/O errors your own way.

The Booting Process 2

Introduction

Pascal 3.0 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 3.0 Workstation System 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 an absolute start address).
- A linked combination of any of the above, produced by the Librarian.

Note that if you compile a program containing two Pascal `MODULE`s, 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 Series 200 computers. 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 internal or 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, then "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 "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, the 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" (ROM stands for Read-Only Memory). 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 68xxx 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 4.0 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" system (resident on a mass storage device) on various mass storage devices such as the built-in minifloppy 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 are used for the nonce. The absolute load-image of the loader was created by 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[1]. 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 I/O 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.

[1] UNIX is a trademark of AT&T Bell Laboratories.

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 in the *Pascal 3.0 Workstation System* manual, in the chapters on the Compiler and Librarian. Once INITLIB is loaded and the installation code executed, the I/O 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 I/O interface cards. The general subject of peripheral configuration is covered in the *Pascal 3.0 Workstation System* manual.

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" (unit table). 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, LAST 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 while LAST is executing.

After TABLE executes, a complete environment exists in which Pascal programs are 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
```

For STARTUP to terminate is usually undesirable.

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 INIT*xxx*, START*xxx* and TABLE*xxx*.

If the file name is SYS*xxxxxxx*, the chosen file names are INIT*xxxxxxx*, START*xxxxxxx* and TABLE*xxxxxxx*. 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.

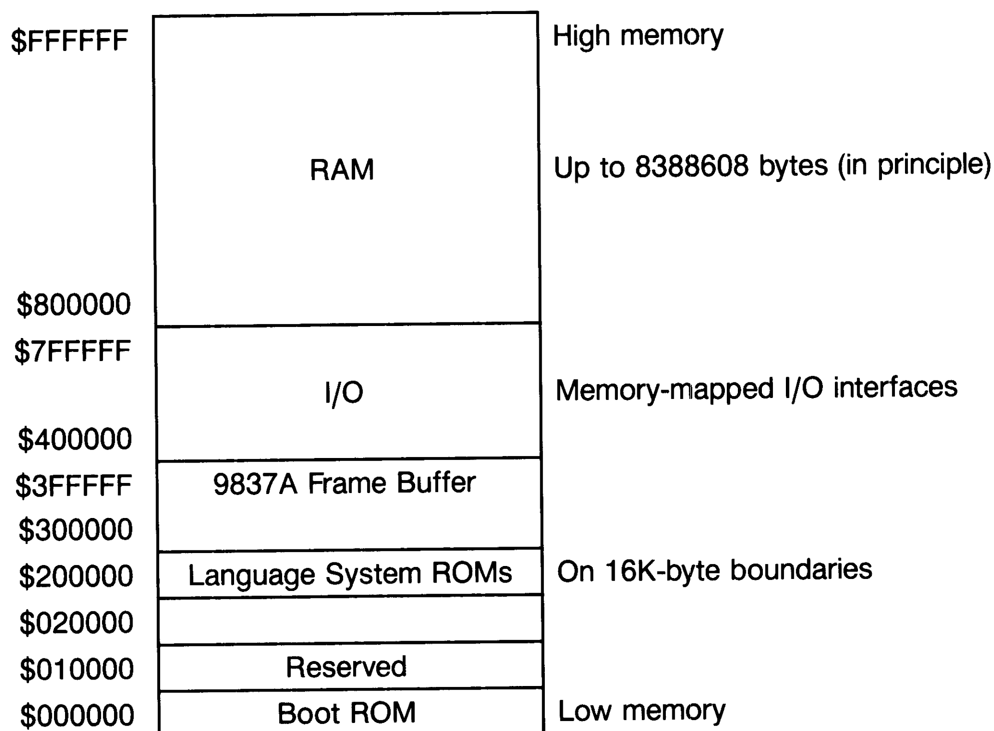
Memory Map Development

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 68xxx 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 68xxx'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 Series 200 machines, the available 16 megabytes are partitioned into areas for ROM, I/O 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. Some Series 200 machines have "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 Series 200 machines are 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 Model 216, 226, 236, 217, and 237 hardware. Pascal 3.0 is designed to run with all of them, but some earlier 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 54K. 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[1]. 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 68xxx is designed 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. Note: interrupt code runs in Supervisor mode, while programs run in User mode, so different stacks are involved.

Series 200 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 six bytes—enough for a long JMP instruction—for each vector.

For instance, at address $00000C (vector 3) the vector content is $FFFFF4, so when an Address Error exception takes place, the CPU will call whatever routine is at $FFFFF4. 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.

[1] Note that the Model 237 uses space from $300000 upward for a frame buffer.

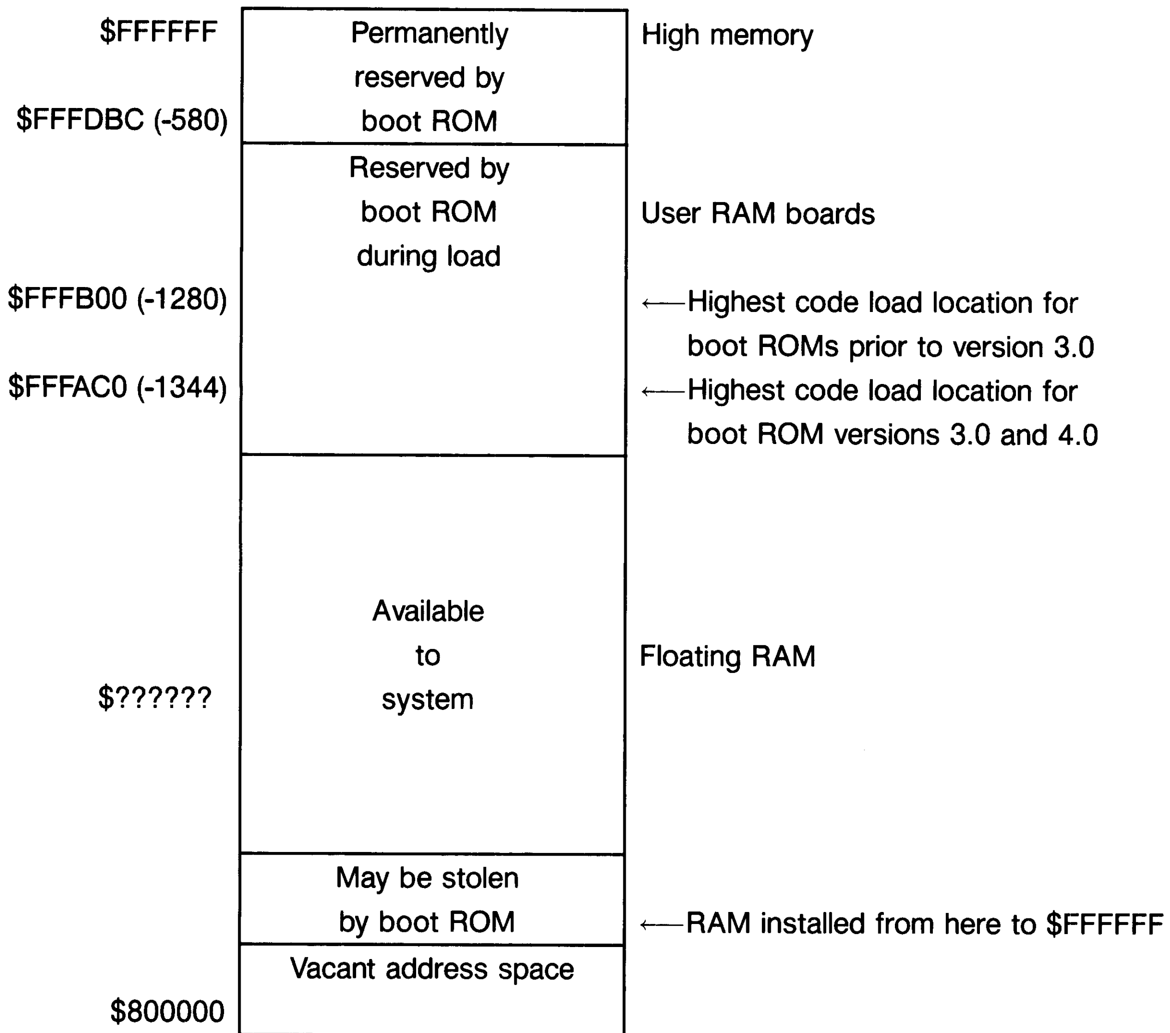
Version 4.0 and 3.0 of the boot ROM used more memory than previous versions. It not only takes more space at the upper end of memory, it also may steal a little at the bottom of physical memory. This only happens when booting from certain specialized devices such as the Shared Resource Manager.

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 `$FFFED4` (−300). Got that?

```
type
  Lrec=                packed record
                         Filler:  packed array [0..19] of char;
                         MemUsed: integer;
                       end;
var
  Thing[-300]:         ^lrec;    {pointer}
       .
       .
       .
BottomBytesStolen:=Thing^.MemUsed;
```

But you should only do this if the machine has the Version 3.0 boot ROM or later. 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.

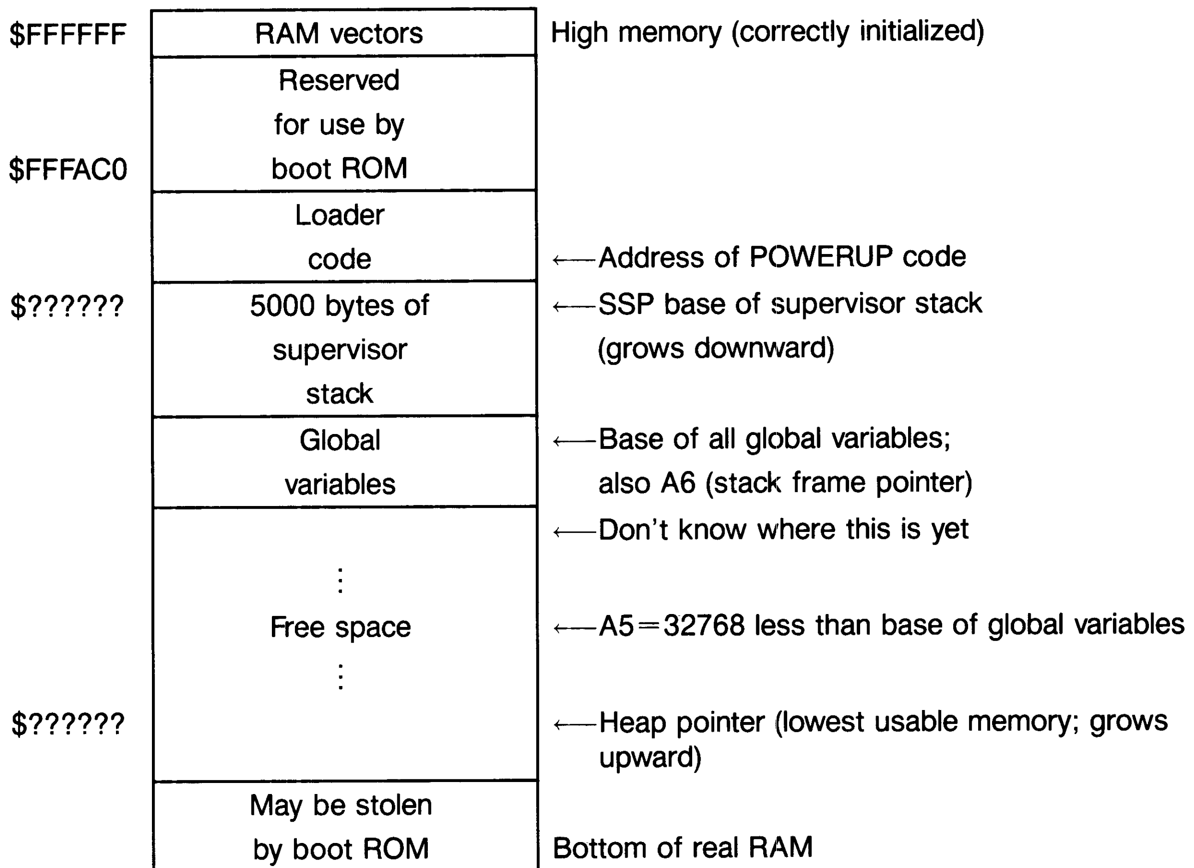
This absolute image is *not* a standard Pascal object code file; it is in a different, much simpler, format. Boot files are generated from linked, relocatable object code by the Librarian's `B` (Boot) command. The loader in the boot ROM is really dumb, so there can be no unsatisfied externals in a boot 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. The Pascal 3.0 boot file is 13 674 bytes long and is loaded at −15400 (`$FFC3D8`).

The boot ROM runs in 68xxx 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:



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 32 768 bytes below the base of the global variable area[2]. 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 −32 768 through 32 767; so the base of the global area is exactly `32766(A5)`.

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

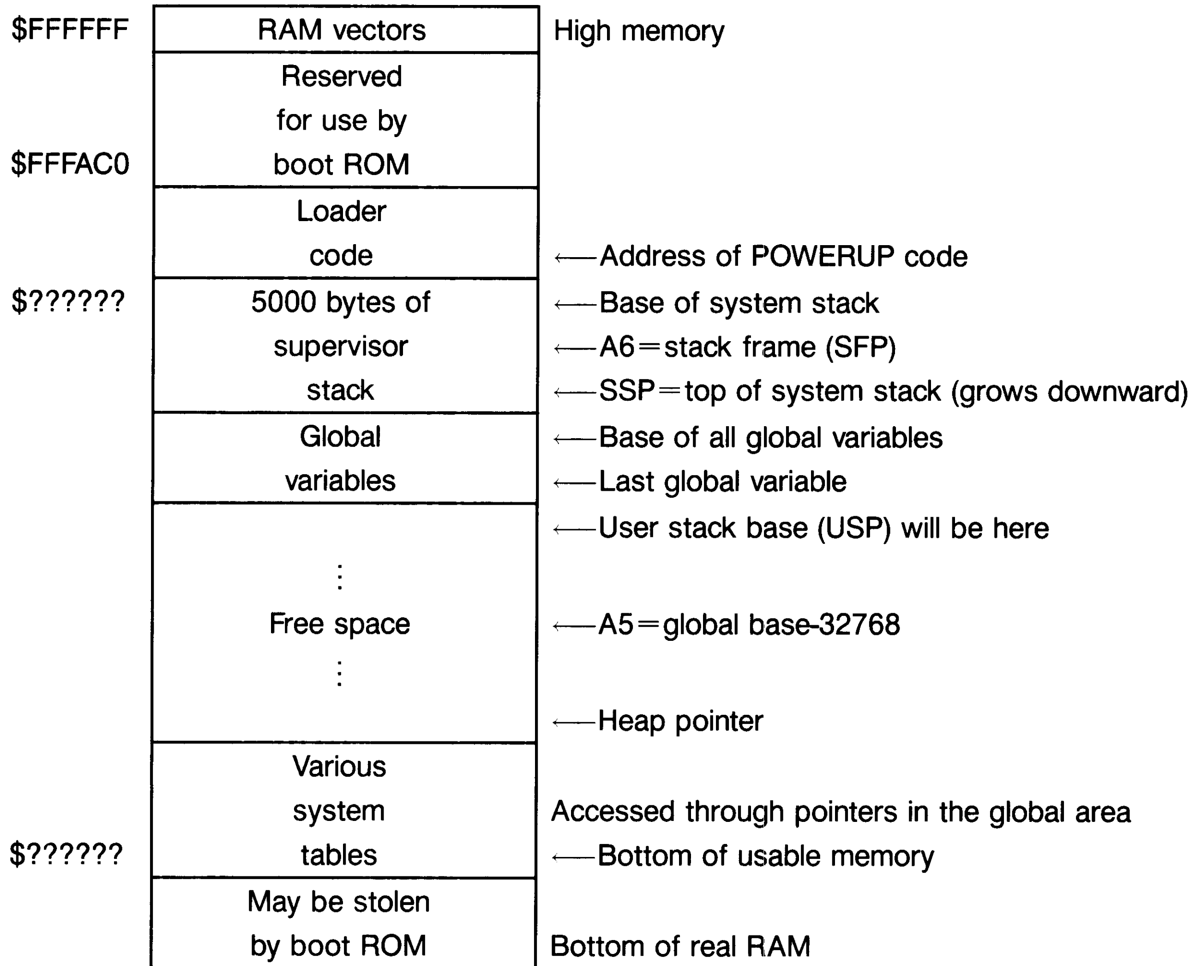
Also, the loader is initialized. This involves finding two pieces of information: how big is the global variable area of the loader itself, and where is the `SYSDEFS` table.

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, i.e., 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 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.

[2] This is a departure from the earlier Pascal 1.0 release, in which **A5** addresses the beginning—the 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 the negative displacements from **A5** were usable. Unfortunately, some of the code in the boot ROM and the BASIC language ROMs assume the convention of the Pascal 1.0 system; so in Pascal 2.0 and later versions, ROM code must be accessed through the special interfacing routine **ROMCALL**.

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



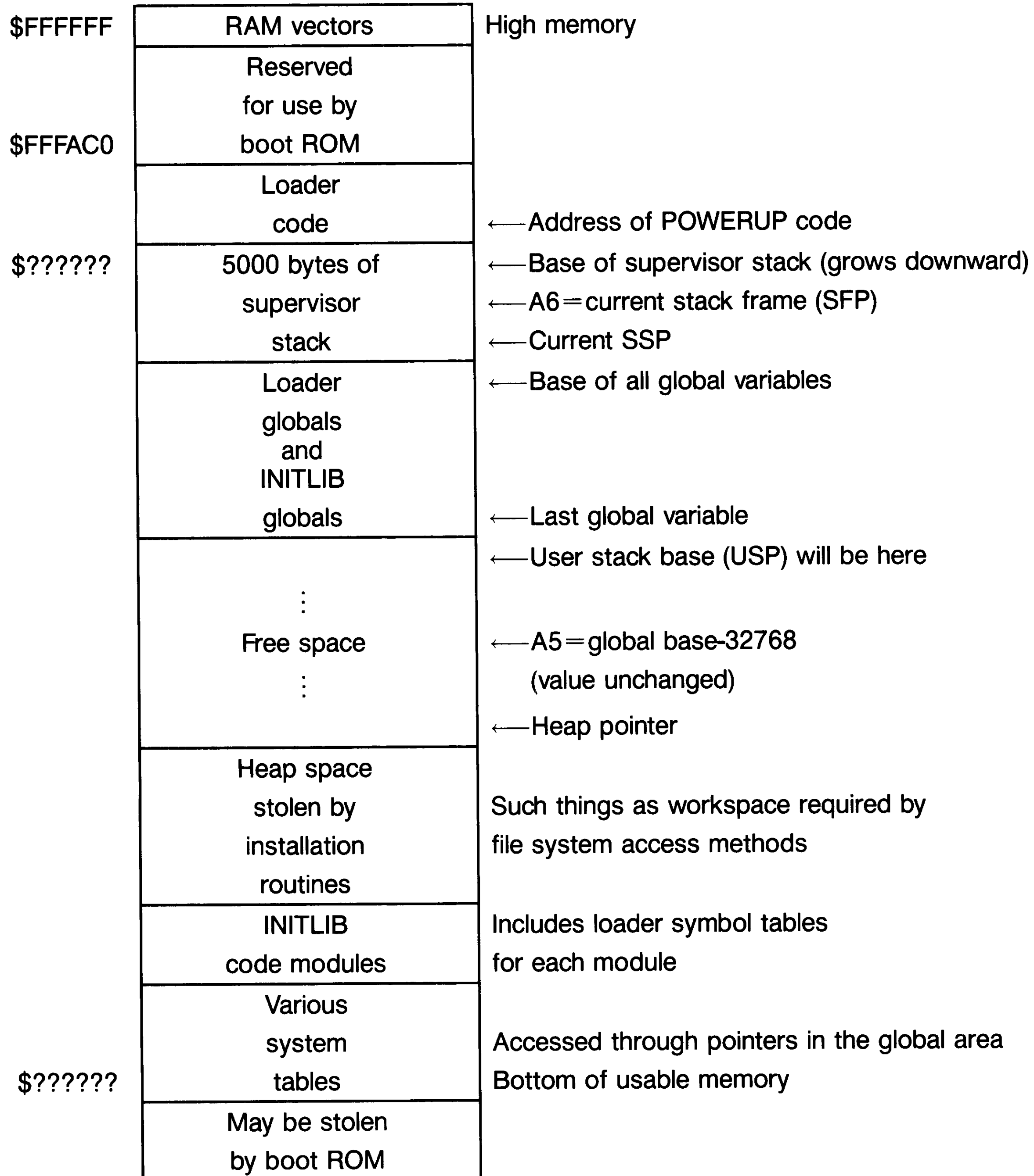
The stack frame pointer (A6) got moved into the system stack area as a side effect of the normal Pascal procedure entry mechanism. It always points to the base of the stack frame for the currently executing procedure.

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 I/O 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 in INITLIB is finished, the system memory map looks like this:



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.

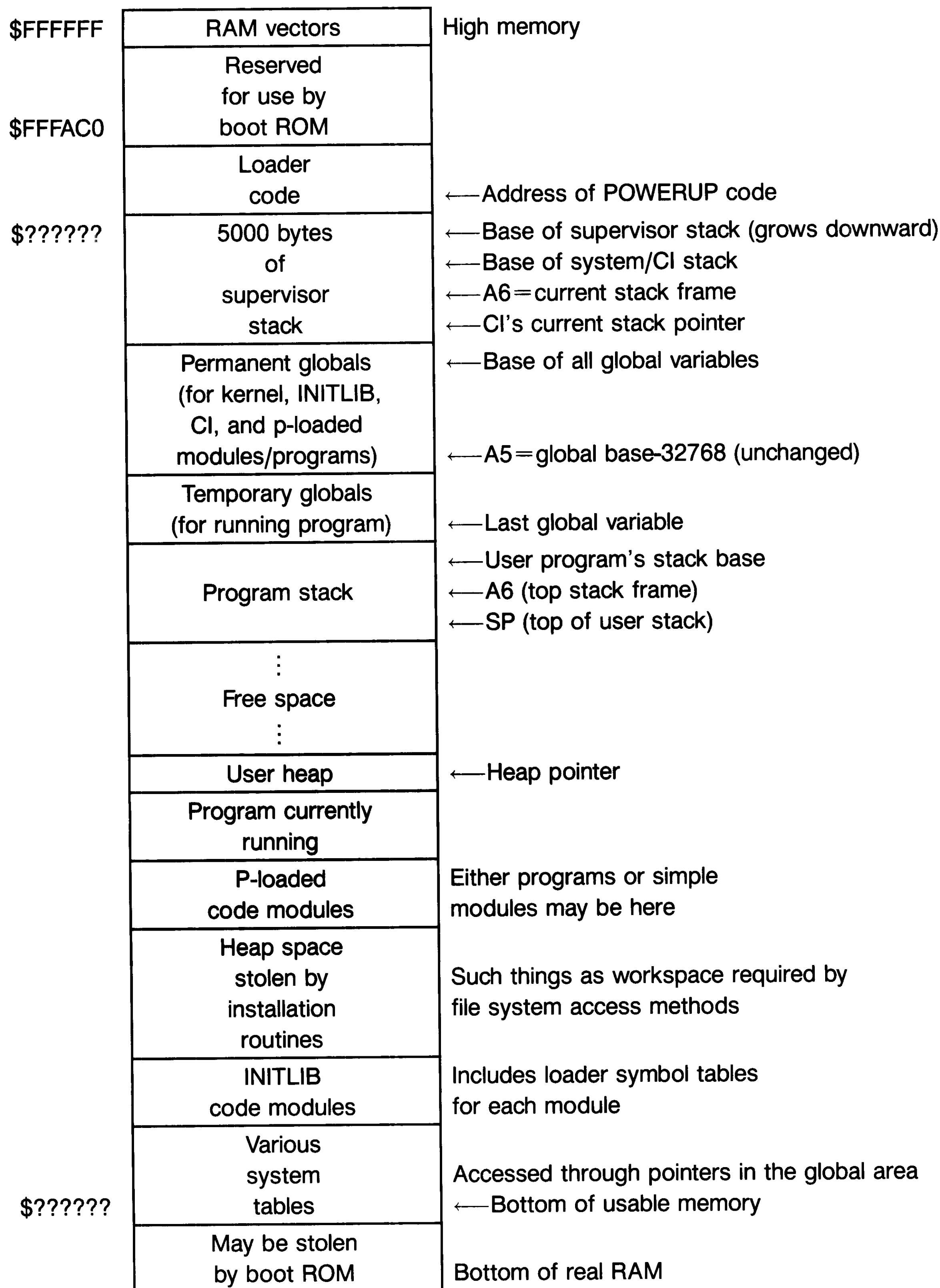
To complete the Pascal boot process, it remains to load the the Command Interpreter program `STARTUP` and the I/O configuration program `TABLE`, and to execute them. If `STARTUP` is found on the boot volume, it is loaded before table; otherwise `STARTUP` is loaded from the system volume after `TABLE` 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 I/O 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 I/O 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.



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 Filer 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 and stream files 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
    Low=          0;
    High=         100;
  type
    Int=          -32768..32767;
    Index=        Low..High;
    Arrae=        array [Index] of Int;
  var                                        {beginning of CHARLIE's globals}
    Head,Tail:    Index;
    List:         Arrae;
  procedure AddToList(K: Int);
  procedure TakeFromList(var K: Int);
implement                      {CHARLIE_CHARLIE (initialization for modules)}
  var
    LastValue:    Int;

  procedure HiddenProc;                           {no symbol if not exported}
  begin
        ....  {body of "HiddenProc" omitted for clarity}
  end;

  procedure AddToList(K: Int);       {CHARLIE_ADDTOLIST: relative to the
                                      beginning of module CHARLIE}
  begin
        ....  {body of "AddToList" omitted for clarity}
  end;

  procedure TakeFromList(var K: Int);  {CHARLIE_TAKEFROMLIST: relative to
                                        the beginning of module CHARLIE}
  begin
        ....  {body of "TakeFromList" omitted for clarity}
  end;
end;        {of module Charlie}
```

Module CHARLIE exports the identifiers LOW, HIGH, 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 the codefile containing 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	*(global VAR address)*
CHARLIE_CHARLIE	*(module initialization)*
CHARLIE_ADDTOLIST	*(exported procedure)*
CHARLIE_TAKEFROMLIST	*(exported procedure)*

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 32 768 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 zeroes only 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[3]. 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.
A804XDVR	Contains entry points for A804XDVR routines SENDCMD, SENDDATA, CMD_READ_1, and others.
KEYS	Code to handle the keyboard and character set mapping. This module exports nothing, it just contains the actual keyboard TM and the ISR code.
NONUSKBD1	No export text. Supports non-U.S. keyboards by setting up character mapping. Handles Katakana for all keyboards.
NONUSKBD2	No export text. Supports non-U.S. keyboards by setting up character mapping.
CRT	No export text. Contains CRT TM code and debugger window handler and sets up character mapping for alpha displays.
CRTB	No export text. Has Pascal and assembly components; contains CRT TM code and debugger window handler for 9837A and sets up character mapping.

[3] Although some INITLIB module listings show export text (e.g., "KEYS" exports "INITKEYS"), in the case of pure drivers, the DEFs have been suppressed during linking, and will not be found in RAM. These pieces of code are inaccessible except via procedure variables if they are currently assigned to the entry points.

`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 elsewhere). Defines the generalized file catalogue entry type.
`MINI`	Driver for built-in minifloppies; calls code in boot ROM.
`INITUNITS`	Generates the initial I/O Unitable setup.
`FS`	File system functions which are called by the Compiler to implement Pascal file I/O. More of this in `INITLIB`.
`SETUPSYS`	Calls initialization routines for modules `POWERUP`, `MISC`, `INITUNITS`, `LDR`, and `FS`.
`SYSDEVS`	This has entry points for clock, beeper, etc., as well as typeahead buffer handlers, and keyboard and CRT TMs.

The dependencies of these modules on each other is shown using the convention that $A \leftarrow B\ C$ 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`	$\leftarrow$	nothing
`POWERUP`	$\leftarrow$	`SYSGLOBALS, LOADER`
`ASM`	$\leftarrow$	`SYSGLOBALS, LOADER`
`LOADER`	$\leftarrow$	`SYSGLOBALS, ASM`
`ISR`	$\leftarrow$	`SYSGLOBALS, ASM`
`MINI`	$\leftarrow$	`SYSGLOBALS, ASM`
`BOOTDAMMODULE`	$\leftarrow$	`SYSGLOBALS, ASM, MINI`
`INITLOAD`	$\leftarrow$	`SYSGLOBALS, ASM, BOOTDAMMODULE, LOADER`

Everything else also uses `SYSGLOBALS` and `ASM`. Moreover,

`CRT`	$\leftarrow$	`SYSGLOBALS, ASM, MISC, SYSDEVS`
`CRTB`	$\leftarrow$	`SYSGLOBALS, ASM, MISC, SYSDEVS`
`KEYS`	$\leftarrow$	`SYSGLOBALS, ASM, MISC, SYSDEVS`
`MISC`	$\leftarrow$	`SYSGLOBALS, ASM,`
`FS`	$\leftarrow$	`SYSGLOBALS, ASM, MISC`
`INITUNITS`	$\leftarrow$	`SYSGLOBALS, ASM, MINI, FS`
`LDR`	$\leftarrow$	`SYSGLOBALS, ASM, MISC, FS, LOADER`
`SETUPSYS`	$\leftarrow$	`SYSGLOBALS, ASM, 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. These loader tables can be used to find the value of (i.e., the 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's symbol tables for the absolute, memory-image kernel. Thus, the loader can link references in 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. These dummy modules are found in the `INTERFACE` file on the `LIB` disc.

The File System 3

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, e.g., 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 3.0 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	The SRM (Shared Resource Manager) directory organization offers 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 that.

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 I/O 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 and later versions of Pascal, and is part of the reason object code is incompatible between these systems.

Note that in this system there is a sharp distinction between file I/O and device I/O. File I/O is provided by the standard statements of Pascal such as `RESET`, `REWRITE`, `GET` and `PUT`. Device I/O 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 I/O 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 I/O 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.

FIBs are Pascal records—complicated objects—whose full description is exported from module `SYSGLOBALS`. Three particularly important fields of a 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 file kinds 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` ⟨*type*⟩"
`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`[1] ("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
FHPOPEN	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 `MFS`)
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

[1] The code is actually in module `REALS` in `INITLIB`.

FWRITEINT	Write one integer
FWRITELN	Write end of line
FWRITEPAOC	Write packed array of char to text
FWRITEREAL	Write real number (in MFS)
FWRITESTR	Write a string
FWRITEWORD	Write a 16-bit integer

Each of these routines requires a FIB as one parameter. See the chapter *File Support* 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.

These are the components of the scalar type "AMREQUESTTYPE":

```
READBYTES
WRITEBYTES
FLUSH
WRITEEOL
READTOEOL
CLEARUNIT
SETCURSOR
GETCURSOR
STARTREAD
STARTWRITE
UNITSTATUS
```

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.

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 a FIBP is a pointer to a FIB, and a WINDOW is an array of bytes. There are several AMs 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 ⟨*type*⟩). In MISC.
TEXTAM	UCSD text file format (skip first 1K bytes, leading blank compression, nulls at end of page). In UCSD_AM.
ASCIIAM	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[2] 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 I/O error by setting IORESULT.

[2] UNIX is a trademark of AT&T Bell Laboratories.

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, e.g., #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	HP-IB primary address, or SRM node address.
UISINTERACTIVE	Indicates whether user can edit input.
UISBLKD	Has a value of TRUE for discs, FALSE for byte-stream devices like printers.
UVID	Name of the volume (if known).
UMEDIAVALID	Media has had files opened on it, and has not been changed since.
UISFIXED	The media is not removable.
UREPORTCHANGES	If false, suppresses messages when drive door opened (the Filer uses this).

Types UNITENTRY and UNITABLETYPE are declared in module SYSGLOBALS. The actual unit table itself resides in operating system heap space 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 TMs ultimately do their work by calling routines available through the Pascal device I/O 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. There is no TMTYPE declaration; AMTYPE is used for both AMs and TMs.

The various TMs 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. TMs 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 a 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, OPENDIRECTORY, 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 a FIB, whereas AMs want a pointer to a FIB. Probably the reasons for this are historic, since passing a pointer by value is the same as passing by reference the object 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 `'*:'`	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.
`'.SYSTM'`	Boot file.
⟨*no suffix*⟩	Pascal "`FILE OF` ⟨*type*⟩".

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( (AM 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 is 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 DAMs make the same choices, but that is a fact rather than a regulation. Here is a table summarizing the choices.

	FKIND		
		Unblocked	**Blocked**
File Type	**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 I/O 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 `EFTTABLE` array can optionally be used to indicate what this external file type is as a short integer. It is not a perfectly general mechanism, since the same file type might require different type designators under different DAMs. 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 a FIB

Refer to the declaration of type FIB exported from module SYSGLOBALS. A 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 I/O 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 (i.e., 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(⟨*type*⟩) in "FILE OF ⟨*type*⟩". The value is zero for UCSD-compatible untyped files. Note that if FRECSIZE equals zero, 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 *How the Access Method is Selected* above.

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, e.g., `REWRITE(F)` as opposed to `REWRITE(F,'CHARLIE')`. Anonymous files will not be `LOCK`ed 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;

This is `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 `FHPOPEN`. 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 I/O", 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 I/O" (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	State Name	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 the 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;

This is 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:

FPOS>0 — Requests BLOCKS×512 bytes, where '[⟨*blocks*⟩]' was appended to the file name.

FPOS=0 — Means no size was specified. Some DAMs will interpret this as a hint to take the largest space available on the disc.

FPOS<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, whichever is larger.

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 I/O 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 end-of-file. 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 Is Selected*).

Note: 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 kluge in the definition of Logical Interchange Format 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 is 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 sixteen 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 sixteen 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; if you really want to do this, you'll have to p-load another copy, and then execute it, or change the name of the code file, and then execute it.

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 local 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, the TM must also add in `FPOS` 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 HP-IB address.

FBUSY: BOOLEAN;
Set TRUE by the TM checking a UNITSTATUS request if an overlapped I/O operation is in progress.

FEFT: SHORTINT;
This is the external file type. Would normally be set to EFTTABLE^[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, or 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 I/O transfer. Presently the only calls which can specify that the I/O 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 "LOCK" or "SAVE". 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..2] OF INTEGER;

Space set aside for future expansion.

FEXTRA2: SHORTINT;

Space set aside for future expansion.

FB0, FB1: BOOLEAN;

Space set aside for future expansion.

Variant Fields

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.

The Unit Table

Refer to the UNITABLE and related types exported from module SYSGLOBALS. Fields of the UNITABLE are initialized by execution of the TABLE configuration program at bootup time.

Pascal file I/O is directed to files which reside on so-called "logical units." The logical unit is a number between one and fifty, called the unit number, which is an index into an array called the UNITABLE. The UNITABLE was mentioned briefly in the File System overview, above. That information is now repeated in more detail.

The logical units of file I/O are to be distinguished from interface "select codes." A select code is a number from zero to 31 from which can be calculated the memory-mapped address of a peripheral's interface circuitry. Be warned that the address calculation is not entirely straightforward. Device I/O is discussed elsewhere.

The logical unit on which a file resides is determined from the "volume name" portion of the file name specification. These forms of volume name are recognized:

(no volume name)	If there is no colon in the file specification, or if there are no characters preceding the colon, the volume name is taken to be the current default volume. Otherwise the character sequence preceding the colon is stripped out and becomes the volume name. This yields one of the next two cases...
* or *:	Either of these is a synonym for the system volume.
#⟨*unit number*⟩	If a notation like #31: is found at the beginning of the file specification, the integer unit number is used directly to index into the UNITABLE. The volume name for the file is taken to be the volume label found on the addressed unit.
⟨*volume name*⟩:	If a notation like MYVOL: is found at the beginning of the file specification, the name is extracted and the UNITABLE entries are searched for a volume of that name. If it is found, the unit number is established; otherwise IORESULT is set to INOUNIT, which is reported as "volume not found".
:⟨*file name*⟩	The volume is the default volume.
*⟨*file name*⟩	The volume is the system volume.
*:⟨*file name*⟩	The volume is the system volume.

A unit number for the FIB is thus established when the file is opened. The unit entry is accessed by the AM and TM, and usually again when the file is closed.

```
const
  maxunit=          50;
type
  unitnum=          0..maxunit;          {zero indicates "no unit"}
  unitentry=        packed record

                        :
                      (fields of unit entry are discussed below.)
                        :

                    end;
  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. Note: 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 I/O 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 64K 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 I/O.

We 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 HP-IB address of the device. Could be used for other purposes by non-HP-IB 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 Model 236 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 HP-IB 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 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 itself keeps track 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 drives `DEVID` contains the actual product number of the device, as returned by the "describe" disc command. For local printers, it 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 sixteen 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, e.g., "`F`" for a 9885 floppy disc. Somewhere the plan got lost, because there are too many kinds of disc.

The Series 200 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
`B`	Bubble memory
`E`	EPROM
`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 3.5″ 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 I/O—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 one character ahead. For instance, unit #1: (CONSOLE: echoing standard input file INPUT) has UISINTERACTIVE true, while unit #2 (SYSTERM: non-echoing standard input file KEYBOARD) doesn't. Thus the predicate EOF(KEYBOARD) will force a one-character look-ahead 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 50—ZMEDIUMCHANGED.) This is used by the Filer to avoid silly messages instructing the operator to swap discs when the medium is not removeable.

UREPORTCHANGE: 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.

File Support 4

Introduction

This section describes 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 I/O actually takes place.

This chapter assumes that you have already read the previous chapter, *File System.* If this is not the case, please read that first.

The sample program below 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. No warnings.
```

The Librarian provided the following disassembly.

```
MODULE    FILEDEMO    Created 24-Oct-84
NOTICE:  (none)
  Produced by Pascal Compiler of  4-Jun-84
  Revision number 3
  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


  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 'O '
     12  0A00            dc.w 2560     or dc.b 10,0       or dc.b '  '
```

```
 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_FWRITEPAOC
     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       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. No warnings.
```

[11] @52

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_FHPOPEN, 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; a length of 0 is perfectly legal.

@78

The Compiler generates a check of the system variable IORESULT, because the program was compiled with the default $IOCHECK ON$. The IORESULT variable is found at SYSGLOBALS-22(A5). If it is 0, the operation was successful; otherwise a TRAP #3 is executed.

[13] @98

To write the value (101-I), the Compiler emits:

- Push address of FIB.
- Compute (101-I) and store in a global, "nameless" variable used for temporary storage. The variable is at global Gbase-682(A5) because this is the main program; in a procedure some local cell would have been used. If you look at the first page of the unassembly, you will see that there are 682 bytes of globals, not, 678 as would be indicated by the compilation listing.
- 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_FHPOPEN:

- Push FIB address.
- Push literal 2, indicating FACCESS = READWRITE.
- Push address of string containing file name.
- Call FS_FHPOPEN.

[19] @222

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 FSEEK.
- 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_FWRITEPAOC (**Write** **P**acked **A**rray **of** **C**haracters)

[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 FIBs. 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 I/O Subsystems

There is a single, simple error reporting mechanism used for errors of file I/O. 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 $IOCHECK ON$ is active (which is the default case), code is emitted after every FS-level call to verify that IORESULT is 0. If it is nonzero, an automatic call ESCAPE(-10) is generated, which will be reported as an I/O error unless it is trapped by TRY/RECOVER somewhere.

The FS, AMs, DAMs, and TMs are all compiled with $IOCHECK OFF$, and must explicitly check for IORESULT 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 the last I/O call.
1	ZBADBLOCK	CRC (Cyclic Redundancy Check) error; the disc could not be successfully read even after several retries.
2	IBADUNIT	Illegal unit number; valid numbers are 1 through 50.
3	ZBADMODE	The TM doesn't know how to do requested transfer.
4	ZTIMEOUT	Device not responding.
5	ILOSTUNIT	Volume name on unit doesn't match any entry in the UNITABLE.
6	ILOSTFILE	The file was purged while open to another FIB.
7	IBADTITLE	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 a file which was already open.
13	INOTOPEN	Tried to close a file which was already closed.
14	IBADFORMAT	Bad input data to formatted numeric read.
15	ZNOSUCHBLK	Attempt to read or write past volume limits.

16	ZNODEVICE	Device is 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 (Direct Memory Access) interface card failed.
23	INOTVALIDSIZE	Specified file size incompatible with 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	The directory is full.
28	ISTROVFL	String bound violation in STRWRITE or STRREAD.
29	IBADCLOSE	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, this means you are using a bad disc).
33	ZNOTREADY	Device not ready.
34	ZNOMEDIUM	No storage medium mounted in drive.
35	INODIRECTORY	No directory or directory not readable by this DAM.
36	IBADFILETYPE	File type designator not recognized by AM.
37	IBADVALUE	Some parameter is 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 I/O 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 (this 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., `STRREAD`, `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.

Pascal Declaration

The Pascal declaration of the routine.

Purpose

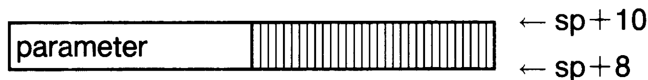
A brief description of what the routine is used for.

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` intruction. Where a symbol similar to this



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`	
`fhpopen`	
`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`	
`fwriteenum`	
`fwriteint`	
`fwriteln`	
`fwritepaoc`	
`fwritereal`	*(mfs)*
`fwritestr`	
`fwritestrbool`	
`fwritestrchar`	
`fwritestrenum`	
`fwritestrint`	*(assembly)*
`fwritestrpaoc`	
`fwritestrreal`	*(mfs)*
`fwritestrstr`	
`fwritestrword`	
`fwriteword`	
`scantitle`	*(function)*
`zapspaces`	

doprefix

Pascal Declaration

```
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

Stack	Offset
	← sp+18
pointer to dirname	← sp+14
pointer to kvid	← sp+10
pointer to kunit	← sp+6
findunit	← sp+4
return address	← stack pointer

Action

1. Call SCANTITLE with DIRNAME.
2. Then call FINDVOLUME.
3. If the unit is found and has a directory, call its DAM with a SETUNITPREFIX request and return the UVID and unit number.
4. If the unit had no directory (specified as #*nn*) and FINDUNIT is FALSE set KVID to #*nn* and return.
5. 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.

fanonfile

Pascal Declaration

```
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

Stack	Offset
	← sp+20
pointer to f	← sp+16
strmax(name)	← sp+14
pointer to name	← sp+10
kind	← sp+8
size	← sp+4
return address	← stack pointer

Action

1. Set FANONYMOUS to TRUE.
2. Call SCANTITLE to extract FVID and FTITLE from name.
3. Set FKIND to KIND.
4. Set FPOS to SIZE.
5. Set FOPTSTRING to ADDR(NULLSTRING) (a dummy value).
6. If FPOS > 0, then set FPOS to FPOS*FBLKSIZE.
7. Set FEFT to EFTTABLE^[FKIND];
8. Set FISNEW to TRUE.
9. Set FREPTCNT to 0.
10. Set FBUFCHANGED to FALSE.
11. Set FLASTPOS to −1.

12. Set FSTARTADDRESS to 0.
13. Set PATHID to -1.
14. Set FNOSRMTEMP to TRUE.
15. Call FINDVOLUME with FVID and FALSE (no verify) to get unit.
16. Call the DAM with a CREATEFILE request.
17. If IORESULT is not INOERROR, call FINDVOLUME with TRUE (do verify) and call DAM with CREATEFILE request again.
18. Set FMODIFIED to TRUE.
19. 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 IORESULT to INOUNIT.
- The DAM may set IORESULT.

fblockio

Pascal Declaration

```
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 (FBLKSIZE) is 512 bytes.

Parameters

F	The FIB.
BUF	The data buffer.
NBLOCKS	The number of blocks to be read or written.
RBLOCK	The starting file position (in blocks) (RBLOCK < 0 indicates current file position).
DOREAD	This is TRUE if reading, FALSE if writing.

Stack

Stack contents	Offset (top of entry)
function result	← sp+26
pointer to f	← sp+22
pointer to buf	← sp+18
nblocks	← sp+14
rblock	← sp+10
doread	← sp+6
return address	← sp+4
	← stack pointer

Action

1. Calculate starting position as follows:
   a. If RBLOCK >= 0, then set FPOS to RBLOCK*FBLKSIZE.
   b. If RBLOCK < 0, then set FPOS to FPOS+(-FPOS) mod FBLKSIZE (i.e., round FPOS to block boundary).
2. Calculate number of bytes to be read/written as follows:
   BLOCKBYTES = NBLOCKS*FBLKSIZE.
3. 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. (In other words, don't attempt to read past end of file. Read to end of file, and return NBLOCKS as number of blocks that include all bytes to end of file. The end of the last block will be uninitialized.)

4. Call the AM to read/write BLOCKBYTES bytes from/to the file at position FPOS.
5. 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.

fbufferref

Pascal Declaration

```
function fbufferref(var f: fib): windowp;
```

Purpose

To return a valid file window.

Parameters

F	The FIB.

Stack

	← sp+12
function result	← sp+8
pointer to f	← sp+4
return address	← stack pointer

Action

1. If `FREADMODE and not FBUFVALID and FLOCKED` then call the AM to read `FRECSIZE` bytes from the file at offset `FPOS` into `FWINDOW^`.
2. If the AM call resulted in an end of file, set `FEOF` 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 a "ghost" end of line).
3. If the AM call did not result in an `EOF`, set `FBUFVALID` to `TRUE` and `FEOF` to `FALSE`. Return `FWINDOW`.
4. If the call to the AM was not necessary (i.e., the *else* for the first *if* above was activated), 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`.

fclose

Pascal Declaration

```
procedure fclose(var f:     fib;
                     ftype: closetype);
```

Purpose

To close a file.

Parameters

F The FIB.

FTYPE The type of close (e.g., CNORMAL, PURGE, LOCK, CCRUNCH).

Stack

	← sp+10
pointer to f	
	← sp+6
ftype	
	← sp+4
return address	
	← stack pointer

Action

1. If the file is not open (not FREADABLE and not FWRITEABLE) don't do anything.
2. If FANONYMOUS or (FTYPE = PURGE) or (FISNEW and (FTYPE = CCRUNCH)), call the DAM with a purgefile request.
3. 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.
4. Then call the DAM with a CLOSEFILE request.
5. 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.

fcloseit

Pascal Declaration

```
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
pointer to f	← sp+8
pointer to stype	← sp+4
return address	← stack pointer

Action

1. 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.
2. Call FCLOSE with the CLOSETYPE constructed.

Errors

- If the file is not open (not FREADABLE and not FWRITEABLE), set IORESULT to INOTOPEN.
- If the STYPE cannot be converted to a CLOSETYPE, set IORESULT to IBADCLOSE.

feof

Pascal Declaration

```
function feof(var f: fib): boolean;
```

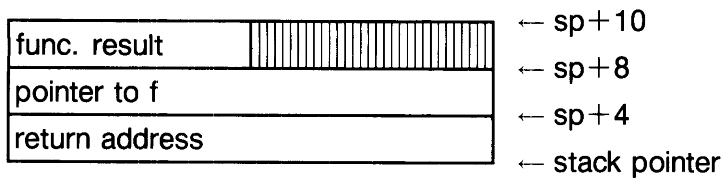
Purpose

To determine if file pointer is at end of file.

Parameters

F	The FIB.

Stack



Action

1. If not (FREADABLE or FWRITEABLE) (i.e., if the file is closed) then return TRUE.
2. If FRECSIZE <= 0 (untyped files) then return TRUE if FPOS > FLEOF, FALSE otherwise.
3. If (FRECSIZE > 0) and FREADABLE and FWRITEABLE then return TRUE if FPOSITION(F) > FMAXPOS(F), FALSE otherwise.
4. 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.

feoln

Pascal Declaration

```
function feoln(var f: fib): boolean;
```

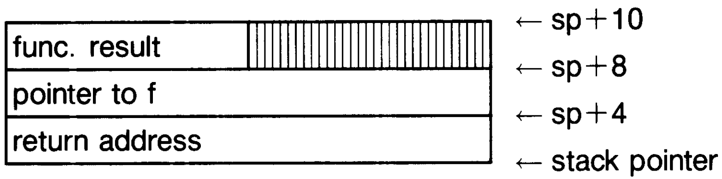
Purpose

To determine if file pointer is at end of line.

Parameters

F   The FIB.

Stack



Action

1. Call FBUFFERREF (to get proper FIB state), set IORESULT to INOERROR if it was IEOF, and return F.FEOLN.

Errors

- FBUFFERREF may set an IORESULT other than IEOF.

fget

Pascal Declaration

```
procedure fget (var f: fib);
```

Purpose

To position file pointer to next record.

Parameters

F	The FIB.

Stack

pointer to f	← sp+8
return address	← sp+4
	← stack pointer

Action

1. If FREADMODE and not FBUFVALID then call FREAD with F and F.FWINDOW^ to get the next record with the AM.
2. 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.

fgetxy

Pascal Declaration

```
procedure fgetxy(anyvar f:     fib;
                    var x,
                        y:     integer);
```

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
pointer to f	← sp+12
pointer to x	← sp+8
pointer to y	← sp+4
return address	← stack pointer

Action

1. Call the AM with a SETCURSOR request.
2. Set X to FXPOS.
3. Set Y to FYPOS.

Errors

- The AM may set IORESULT (e.g., to IBADREQUEST).

fgotoxy

Pascal Declaration

```
procedure fgotoxy(anyvar f:     fib;
                         x,
                         y:     integer);
```

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
pointer to f	← sp+12
x	← sp+8
y	← sp+4
return address	← stack pointer

Action

1. Set FXPOS to X.
2. Set FYPOS to Y.
3. Call the AM with a SETCURSOR request.

Errors

- The AM may set IORESULT (e.g., to IBADREQUEST).

fhpopen

Pascal Declaration

```
procedure fhpopen(var f:        fib;
                      typ:      faccess;
                  var title,
                      option:   string255);
```

Purpose

To open a file with a name and an optional third parameter (Pascal RESET, OPEN, APPEND, REWRITE procedures); for example, RESET(F,TITLE,OPTION).

Parameters

F	The FIB.
TYP	The type of open (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

Stack	Offset
	← sp+18
pointer to f	← sp+14
typ	← sp+12
pointer to title	← sp+8
pointer to option	← sp+4
return address	← stack pointer

Action

1. Call FCLOSE with F and CNORMAL.
2. Set FANONYMOUS to FALSE.
3. Call SCANTITLE to extract FVID and FTITLE and FKIND and FILESIZE from name.
4. Set FPOS to the FILESIZE extracted.
5. Set FOPTSTRING to ADDR(OPTION).
6. If FPOS > 0 then set FPOS to FPOS*FBLKSIZE.
7. Set FEFT to EFTTABLE^[FKIND].
8. If TYP = WRITEONLY, set FISNEW to TRUE, FALSE otherwise.
9. Set the following variables to these values:

```
freptcnt        0
fbufchanged     false
flastpos        -1
fstartaddress   0
pathid          -1
fnosrmtemp      true
```

10. For TYP = READONLY (RESET) do
    a. Call FINDVOLUME with FVID and FALSE (no verify) to get unit.
    b. Call the DAM with a OPENFILE request.
    c. If IORESULT is not INOERROR, call FINDVOLUME with TRUE (do verify) and call DAM with OPENFILE request again.
11. For TYP = READWRITE (OPEN) and for TYP = WRITEAPPEND (APPEND) do:
    a. Call FINDVOLUME with FVID and FALSE (no verify) to get unit.
    b. Call the DAM with an OPENFILE request.
    c. If IORESULT is not INOERROR and not INOFILE, call FINDVOLUME with TRUE (do verify) and call DAM with OPENFILE request again.
    d. If it fails again, set FISNEW to TRUE (revert to FMAKETYPE).
    e. If OPENFILE request succeeds, then if FPOS > FPEOF, call the DAM with a STRETCHIT request.
    f. If the STRETCHIT request fails (if FPOS still > FPEOF), set IORESULT to ICANTSTRETCH.
    g. If FISNEW (reverted to FMAKETYPE), 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.
    h. Now that the hard stuff is done, if TYP is READWRITE and FISTEXTVAR is TRUE, call the DAM to close the file and set IORESULT to IBADFILETYPE (not allowed to OPEN or APPEND text files).
12. For TYP=WRITEONLY (REWRITE), do:
    a. Call FINDVOLUME with FVID and FALSE (no verify) to get unit.
    b. Call the DAM with a CREATEFILE request.
    c. If IORESULT is not INOERROR, call FINDVOLUME with TRUE (do verify) and call DAM with CREATEFILE request again.
13. Now for all values of TYP set FMODIFIED to FISNEW.

14. 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.

fhpreset

Pascal Declaration

```
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 (READONLY = RESET; READWRITE = OPEN; WRITEAPPEND = APPEND; WRITEONLY = REWRITE.)

Stack

pointer to f	← sp+10
typ	← sp+6
return address	← sp+4
	← stack pointer

Action

1. If the file is not already open, it must be created as follows.
   a. Set the following variables to these values:

fanonymous	true
fvid	the system volume
fkind	datafile
fpos	-1 (half the largest or second largest space, whichever is largest)
foptstring	0-length string (a dummy value)
feft	efttable^[fkind]
fisnew	true
freptcnt	0
fbufchanged	false
flastpos	-1
fstartaddress	0
pathid	-1
fnosrmtemp	true

   b. Call FINDVOLUME with FVID and FALSE (no verify) to get unit.
   c. Call the DAM with a CREATEFILE request.
   d. If IORESULT is not INOERROR, call FINDVOLUME with TRUE (do verify) and call DAM with CREATEFILE request again.
   e. Set FMODIFIED to TRUE.
2. 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 IORESULT to IBADTITLE.
- If FINDVOLUME returns unit 0, set IORESULT to INOUNIT.
- The DAM may set IORESULT.

findvolume

Pascal Declaration

```
function findvolume (var fvid:        vid;
                         verify:      boolean): unitnum;
```

Purpose

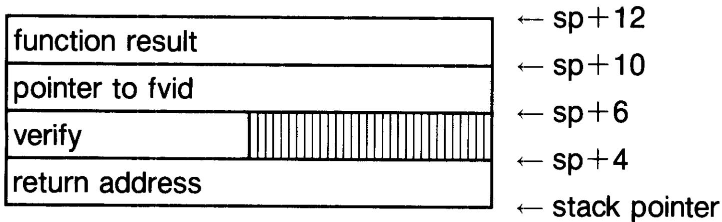
To find the unit associated with the volume id FVID. With VERIFY TRUE or with FVID of the form "#*nn*" FVID is also set to the actual volume name.

Parameters

FVID The volume id.

VERIFY Boolean to indicate whether DAM must be called if searching by name.

Stack



Action

1. 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*.
2. If FVID is not of the form #*nn* then
   a. Search the unit table (unit 50 to unit 1 since faster devices such as hard discs are usually assigned to higher unit numbers) for UVID = FVID (uppercased if unit entry so indicates).
   b. 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.
   c. 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.

finitb

Pascal Declaration

```
procedure finitb(var f:          fib;
                     window:     windowp;
                     recbytes:   integer);
```

Purpose

To initialize a 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
pointer to f	← sp+12
window	← sp+8
recbytes	← sp+4
return address	← stack pointer

Action

1. 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
```

2. Set FWINDOW to WINDOW.
3. Set FISTEXTVAR to TRUE if RECBYTES = −3, FALSE otherwise.
4. If RECSIZE = −1 (untyped file), then set FWINDOW to nil and set FRECSIZE to 0.
5. If RECSIZE <= 0 (except −1) set FRECSIZE to 1 and initialize the first character of FWINDOW^ to chr(0).
6. Otherwise, set FRECSIZE to RECBYTES.
7. Set FBUFFERED to TRUE if FRECBYTES > 0, and FALSE otherwise.

Errors

None.

fixname

Pascal Declaration

```
procedure fixname(var title:     string;
                      kind:      filekind);
```

Purpose

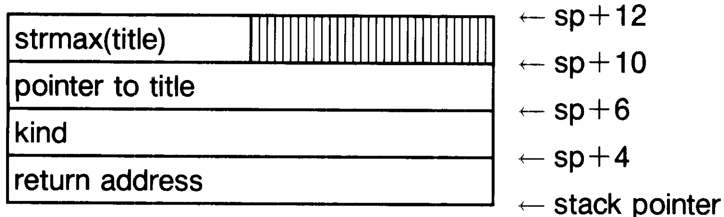
To put proper suffixes on file names. Also removes spaces and control characters.

Parameters

TITLE The file name to be fixed.

KIND The file type associated with the suffix.

Stack



Action

1. Call ZAPSPACES with TITLE.
2. If TITLE ends in ':', then do nothing.
3. If TITLE ends in '.' then remove last character.
4. Otherwise, if a call to suffix with TITLE returns DATAFILE (i.e., no suffix is already present) then look up the 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.

fmaketype

Pascal Declaration

```
procedure fmaketype(anyvar f:              fib;
                       var title,
                           option,
                           typestring:     string);
```

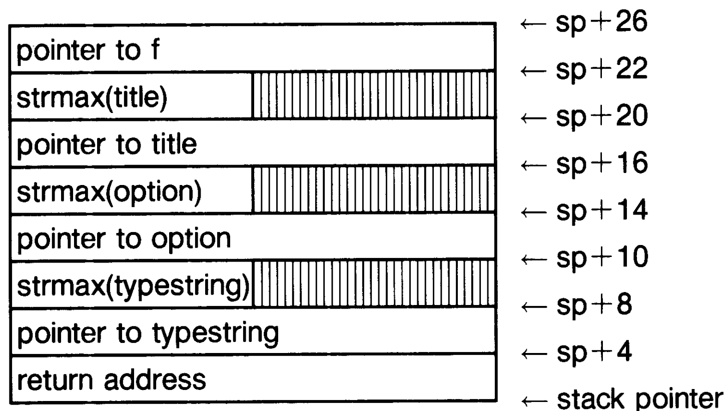
Purpose

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



Action

1. Call FCLOSE with F and CNORMAL.
2. Set FANONYMOUS to FALSE.
3. Call SCANTITLE to extract FVID and FTITLE and FILESIZE from name.
4. Set FPOS to the FILESIZE extracted.
5. Call SUFFIX with TYPESTRING and set FKIND to the kind returned.
6. Set FOPTSTRING to ADDR(OPTION).
7. If FPOS > 0, then set FPOS to FPOS*FBLKSIZE.
8. Set the following variables to these values:

```
feft            efttable^[FKIND]
fisnew          true
freptcnt        0
fbufchanged     false
flastpos        -1
fstartaddress   0
pathid          -1
fnosrmtemp      true
```

9. Call FINDVOLUME with FVID and FALSE (no verify) to get unit.
10. Call the DAM with a CREATEFILE request.
11. If IORESULT is not INOERROR, call FINDVOLUME with TRUE (do verify) and call DAM with CREATEFILE request again.
12. Set FMODIFIED to TRUE.
13. 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.

fmaxpos

Pascal Declaration

```
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
pointer to f	← sp+4
return address	← stack pointer

Action

1. Return FLEOF div FRECSIZE.

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.

foverfile

Pascal Declaration

```
procedure foverfile (anyvar f:           fib;
                        var title,
                            option,
                            typestring:   string);
```

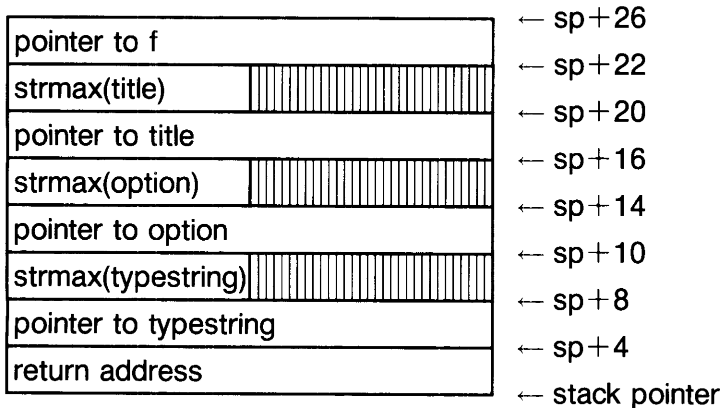
Purpose

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



Action

1. Call FCLOSE with F and CNORMAL.
2. Set FANONYMOUS to FALSE.
3. Call SCANTITLE to extract FVID and FTITLE and FILESIZE from NAME.
4. Set FPOS to the FILESIZE extracted.
5. Call SUFFIX with TYPESTRING and set FKIND to the kind returned.
6. Set FOPTSTRING to ADDR(OPTION).
7. If FPOS > 0, then set FPOS to FPOS*FBLKSIZE.
8. Set the following variables to these values:

```
feft              efttable^[FKIND]
fisnew            false
freptcnt          0
fbufchanged       false
flastpos          -1
fstartaddress     0
pathid            -1
fnosrmtemp        true
```

9. Call FINDVOLUME with FVID and FALSE (no verify) to get unit.
10. Call the DAM with an OVERWRITEFILE request.
11. If (IORESULT <> INOERROR) and (IORESULT <> INOFILE), call FINDVOLUME with TRUE (do verify) and call DAM with OVERWRITEFILE request again.
12. If it fails again, set FISNEW to TRUE (revert to FMAKETYPE).
13. If OVERWRITEFILE request succeeds, then if FPOS > FPEOF, call the DAM with a STRETCHIT request.
14. 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.
15. If FISNEW (reverted to FMAKETYPE), 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.
16. Set up file state as follows.

```
fmodified       true
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.

foverprint

Pascal Declaration

```
procedure foverprint(var t: text);
```

Purpose

To write an overprint command to a text file.

Parameters

`T`	The text file.

Stack

	← sp+8
pointer to t	← sp+4
return address	← stack pointer

Action

1. Write an eol (carriage return—`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.

fpage

Pascal Declaration

```
procedure fpage (var t: text);
```

Purpose

To write a page eject sequence to a text file.

Parameters

T	The text file.

Stack

	← sp+8
pointer to t	
	← sp+4
return address	
	← stack pointer

Action

1. Write an eol (carriage return—`chr(13)`) and clear scr (formfeed—`chr(12)`) to the text file.

Errors

None.

fposition

Pascal Declaration

```
function fposition (var f: fib): integer;
```

Purpose

To determine the position of the file pointer.

Parameters

F	The FIB.

Stack

Stack	
	← sp+12
function result	← sp+8
pointer to fib	← sp+4
return address	← stack pointer

Action

1. Return FPOS div FRECSIZE + 1 - ord(FBUFVALID).

Errors

- Return 0 on all errors.
- If not (FREADABLE or FWRITEABLE), set IORESULT to INOTOPEN.
- If not FLOCKED, set IORESULT to IFILEUNLOCKED.

fput

Pascal Declaration

```
procedure fput (var f: fib);
```

Purpose

To write the file window to the file.

Parameters

F	The FIB.

Stack

Stack	
	← sp+8
pointer to f	← sp+4
return address	← stack pointer

Action

1. Call FWRITE with F and F.FWINDOW^.

Errors

See FWRITE.

fread

Pascal Declaration

```
procedure fread(anyvar f:     fib;
                anyvar buf:   window);
```

Purpose

To read a record from the file.

Parameters

F — The FIB.

BUF — The buffer into which to read the record.

Stack

	← sp+12
pointer to f	← sp+8
pointer to buf	← sp+4
return address	← stack pointer

Action

1. If FBUFVALID and FLOCKED then move FRECSIZE 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 FRECSIZE bytes into BUF from the file at position FPOS.
2. Set lazy I/O condition by setting FREADMODE to TRUE and FBUFVALID to FALSE.
3. If the AM call resulted in an end-of-file 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.

freadbool

Pascal Declaration

```
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
pointer to t	← sp+8
pointer to b	← sp+4
return address	← stack pointer

Action

1. Call FREADENUM with the address of a constant table of string values for the enumerated type (FALSE,TRUE).
2. If the index (scalar) returned is 1, set B to TRUE. Otherwise set B to FALSE.

Errors

- FREADENUM may set IORESULT.

freadbytes

Pascal Declaration

```
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
pointer to f	← sp+12
pointer to buf	← sp+8
size	← sp+4
return address	← stack pointer

Action

1. Call the AM with a READBYTES request to read SIZE bytes into BUF from the file at position FPOS.
2. 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.

freadchar

Pascal Declaration

```
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

pointer to t	← sp+12
pointer to ch	← sp+8
return address	← sp+4
	← stack pointer

Action

1. Call FREAD with T and CH.

Errors

See FREAD.

freadenum

Pascal Declaration

```
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

pointer to t	← sp+16
pointer to i	← sp+12
p	← sp+8
return address	← sp+4
	← stack pointer

Action

1. 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 (maximum of 255 characters).
2. 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 IORESULT.
- Get (compiler-generated call to FGET) may set IORESULT.
- FREADSTRENUM may set IORESULT or escape on failures.

freadint

Pascal Declaration

```
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
pointer to t	← sp+8
pointer to i	← sp+4
return address	← stack pointer

Action

1. 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 (+ or -) and all digit characters ('0' to '9') up to the next non-digit into a string (maximum of 255 characters).
2. Call STRREAD (compiler-generated call to FREADSTRINT) to convert the constructed string to an integer (I).

Errors

- T^ (compiler-generated call to FBUFFERREF) may set IORESULT.
- Get (compiler-generated call to FGET) may set IORESULT.
- FREADSTRINT may set IORESULT or escape on failures.

freadln

Pascal Declaration

```
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

pointer to t	← sp+8
return address	← sp+4
	← stack pointer

Action

1. While the text file is not at end-of-line, do `GET`s on the text file to skip characters (handle backspace and clearline if unit is interactive).
2. Do one more `GET` on the text file to consume the end-of-line marker.

Errors

None.

freadpaoc

Pascal Declaration

```
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
pointer to t	← sp+10
pointer to a	← sp+6
aleng	← sp+4
return address	← stack pointer

Action

1. 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.
2. If the unit is not interactive (no need to read one character at a time to handle backspace and clearline), read and save the lookahead 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 lookahead character (see above) in A[1]. This may seem roundabout, but much of the code is also used in reading strings.
3. Fill the rest of the array with spaces.

Errors

- T^ (compiler-generated call to FBUFFERREF) may set IORESULT.
- Get (compiler-generated call to FGET) may set IORESULT.
- AM may set IORESULT.

freadreal

Pascal Declaration

```
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
pointer to t	
	← sp+8
pointer to x	
	← sp+4
return address	
	← stack pointer

Action

1. 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 (i.e., sign, digits, decimal point, exponent field) up to the next invalid character into a string (maximum of 255 characters).
2. Call STRREAD (compiler-generated call to FREADSTRREAL) to convert the constructed string to a real (X).

Errors

- T^ (compiler-generated call to FBUFFERREF) may set IORESULT.
- Get (compiler-generated call to FGET) may set IORESULT.
- If T is at end-of-file, set IORESULT to IEOF.
- FREADSTRREAL may set IORESULT on failures.

freadstr

Pascal Declaration

```
procedure freadstr(var t: text;
                   var s: string);
```

Purpose

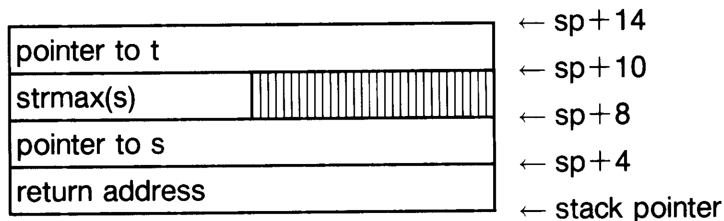
To read (formatted) a string from a text file.

Parameters

T The text file.

A The string to be read into.

Stack



Action

1. 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.
2. If the unit is not interactive (no need to read one character at a time to handle backspace and clearline), read and save the lookahead 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 lookahead character (see above) in `S[1]`. This may seem roundabout 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`.

freadstrbool

Pascal Declaration

```
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
pointer to s	
	← sp+12
pointer to p2	
	← sp+8
pointer to b	
	← sp+4
return address	
	← stack pointer

Action

1. Call FREADSTRENUM with the address of a constant table of string values for the enumerated type (FALSE,TRUE).
2. If the index (scalar) returned is 1, set B to TRUE. Otherwise set B to FALSE.

Errors

- FREADSTRENUM may set IORESULT.

freadstrchar

Pascal Declaration

```
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
pointer to s	← sp+12
pointer to p2	← sp+8
pointer to ch	← sp+4
return address	← stack pointer

Action

1. Set CH to S[P2].
2. Increment P2 by 1.

Errors

- If (P2 < 1) or (P2 > strlen(S)), set IORESULT to ISTROVFL.

freadstrenum

Pascal Declaration

```
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
pointer to s	← sp+16
pointer to p2	← sp+12
pointer to i	← sp+8
p	← sp+4
return address	← stack pointer

Action

1. 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 (maximum of 80 characters).
2. Add the number of characters skipped and copied to P2.
3. Search the table P for the identifier and set I to the index.

Errors

- If (P2 < 1) or (P2 > strlen(S)), set IORESULT to ISTROVFL.
- If the leading spaces extend to the end of S, set IORESULT to ISTROVFL.
- If the identifier does not start with 'A' to 'Z' or 'a' to 'z', set IORESULT to IBADFORMAT.
- If the identifier is not found in the table P, set IORESULT to IBADFORMAT.

freadstrint

Pascal Declaration

```
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
pointer to s	← sp+12
pointer to p2	← sp+8
pointer to i	← sp+4
return address	← stack pointer

Action

This routine is written in assembly for speed.

1. Starting at S[P2], skip spaces.
2. If the first non-space is a sign, remember it.
3. Initialize an accumulator value to 0.
4. 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.
5. Add the number of characters skipped and used to P2.
6. Adjust the sign of the accumulator and assign it to I.

Errors

- If (P2 < 1) or (P2 > strlen(S)), set IORESULT to IBADFORMAT.
- If not at least one digit, set IORESULT to IBADFORMAT.
- If number is too large (overflow), set IORESULT to IBADFORMAT.

freadstrpaoc

Pascal Declaration

```
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
pointer to s	← sp+14
pointer to p2	← sp+10
pointer to a	← sp+6
aleng	← sp+4
return address	← stack pointer

Action

1. Initialize A to all spaces.
2. 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.
3. Add the number of characters copied to P2.

Errors

- If (P2 < 1) or (P2 > strlen(S)), set IORESULT to ISTROVFL.

freadstrreal

Pascal Declaration

```
procedure freadstrreal(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
pointer to s	← sp+12
pointer to p2	← sp+8
pointer to x	← sp+4
return address	← stack pointer

Action

1. Starting at S[P2], skip spaces.
2. Read all valid characters for a real number (e.g., sign, digits, decimal point, exponent field) up to the next invalid character.
3. Convert the characters read into a real (X).
4. Set P2 to one past the last character read.

Errors

- If (P2 < 1) or (P2 > strlen(S)), set IORESULT to ISTROVFL.
- If there is no valid real represented by the characters, set IORESULT to IBADFORMAT.

freadstrstr

Pascal Declaration

```
procedure freadstrstr(var t:     string255;
                      var p2:    integer;
                      var s:     string);
```

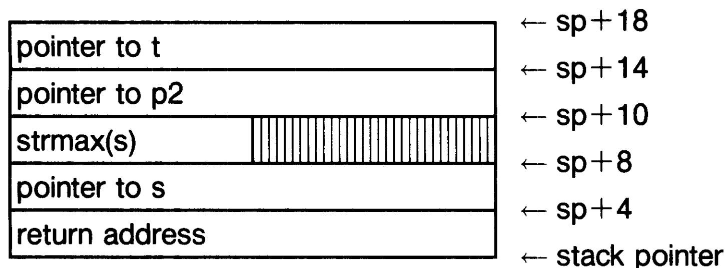
Purpose

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



Action

1. 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.
2. Add the number of characters copied to P2.

Errors

- If (P2 < 1) or (P2 > strlen(T)), set IORESULT to ISTROVFL.

freadstrword

Pascal Declaration

```
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
pointer to s	
	← sp+12
pointer to p2	
	← sp+8
pointer to i	
	← sp+4
return address	
	← stack pointer

Action

1. Call `STRREAD` (compiler-generated call to `FREADSTRINT`) to read an integer from the string.
2. If the integer is not in the range −32 768 to 32 767 then `ESCAPE(-8)`. Otherwise, set `I` to the integer read.

Errors

- `FREADSTRINT` may set `IORESULT` or escape on failures.
- If the integer is not in range, `ESCAPE(-8)`.

freadword

Pascal Declaration

```
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
pointer to t	← sp+8
pointer to i	← sp+4
return address	← stack pointer

Action

1. Read (compiler-generated call to FREADINT) an integer (32-bit) from T.
2. If the integer is not in the range −32 768 to 32 767 then ESCAPE(-8). Otherwise, set I to the integer read.

Errors

- The integer is out of range: ESCAPE(-8).
- FREADINT may set IORESULT.

fseek

Pascal Declaration

```
procedure fseek(var f:        fib;
                    position:  integer);
```

Purpose

To reposition the file pointer.

Parameters

F	The FIB.
FPOSITION	The desired record position of the pointer.

Stack

	← sp+12
pointer to f	
	← sp+8
position	
	← sp+4
return address	
	← stack pointer

Action

1. If FPOSITION < 1, then set FPOS to 0. Otherwise, set FPOS to (POSITION-1)*FRECSIZE.
2. 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.

fstripname

Pascal Declaration

```
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

Stack contents	Offset
pointer to s	← sp+26
strmax(pvname)	← sp+22
pointer to pvname	← sp+20
strmax(ppath)	← sp+16
pointer to ppath	← sp+14
strmax(pfname)	← sp+10
pointer to pfname	← sp+8
return address	← sp+4
	← stack pointer

Action

1. SCANTITLE is called with the FID passed in.
2. Then FINDVOLUME is called to find the volume indicated by the FID.
3. 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 INOUNIT.
- Otherwise, the DAM may set the IORESULT as appropriate (e.g., IBADTITLE, etc.).

fwrite

Pascal Declaration

```
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
pointer to f	← sp+8
pointer to buf	← sp+4
return address	← stack pointer

Action

1. Call the AM with a WRITEBYTES request to write FRECSIZE bytes from BUF at position FPOS.
2. Set 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 FWRITEABLE, set IORESULT to INOTWRITEABLE.
- If not FLOCKED, set IORESULT to IFILEUNLOCKED.
- The AM may set IORESULT.

fwritebool

Pascal Declaration

```
procedure fwritebool(var t:     text;
                         b:     boolean;
                         rleng: shortint);
```

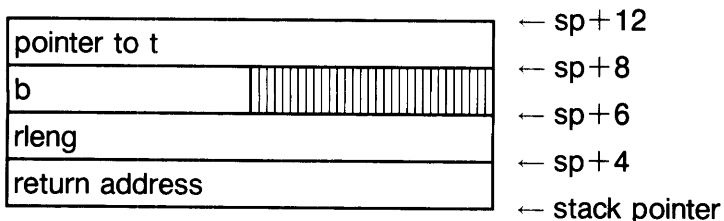
Purpose

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 (maximum 255).

Stack



Action

1. 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 IORESULT.

fwritebytes

Pascal Declaration

```
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
pointer to f	← sp+12
pointer to buf	← sp+8
size	← sp+4
return address	← stack pointer

Action

1. Call the AM with a WRITEBYTES request to write SIZE bytes from BUF at position FPOS.
2. Set 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 FWRITEABLE, set IORESULT to INOTWRITEABLE.
- If not FLOCKED, set IORESULT to IFILEUNLOCKED.
- The AM may set IORESULT.

fwritechar

Pascal Declaration

```
procedure fwritechar(var t:     text;
                         ch:    char;
                         rleng: shortint);
```

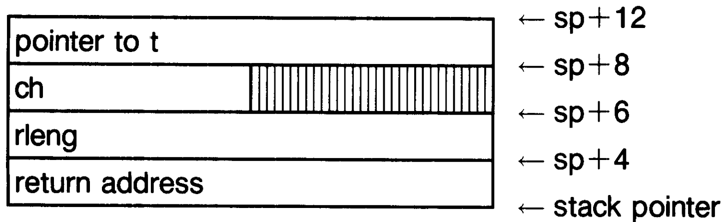
Purpose

To write (formatted) a character to a text file.

Parameters

T	The text file.
CH	The character to be written.
RLENG	The field width to be written into (maximum 255).

Stack



Action

1. If RLENG < 1 then set RLENG to 1.
2. Construct a packed array of characters of RLENG-1 spaces followed by the character CH.
3. Call FWRITEBYTES to write RLENG characters from the packed array to the text file.

Errors

- RLENG > 255 will cause boundary error.
- FWRITEBYTES may set IORESULT.

fwriteenum

Pascal Declaration

```
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 (maximum 255).
P	The compiler generated table of string values for the enumerated type.

Stack

Stack contents	Offset
	← sp+16
pointer to t	← sp+12
i	← sp+10
rleng	← sp+8
p	← sp+4
return address	← stack pointer

Action

1. Call FWRITESTRENUM to write the identifier to a string (i.e., to do the hard part).
2. If IORESULT is INOERROR then call FWRITEBYTES to write the string to the file.

Errors

- FWRITESTRENUM may set IORESULT.
- FWRITEBYTES may set IORESULT.

fwriteint

Pascal Declaration

```
procedure fwriteint(var t:      text;
                        i:      integer;
                        rleng:  shortint);
```

Purpose

To write (formatted) an integer to a text file.

Parameters

T	The text file.
I	The integer to be written.
RLENG	The field width to be written into (maximum 255).

Stack

	← sp+14
pointer to t	
	← sp+10
i	
	← sp+6
rleng	
	← sp+4
return address	
	← stack pointer

Action

1. Call STRWRITE (compiler-generated call to FWRITESTRINT) to write the integer to a string.
2. If IORESULT is INOERROR, call FWRITEBYTES to write the string to the text file.

Errors

- FWRITESTRINT may set IORESULT.
- FWRITEBYTES may set IORESULT.

fwriteln

Pascal Declaration

```
procedure fwriteln(var f: fib);
```

Purpose

To write an end of line marker to the file.

Parameters

F	The FIB.

Stack

pointer to f	← sp+8
return address	← sp+4
	← stack pointer

Action

1. Call the AM with a WRITEEOL request at position FPOS.
2. Set 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 FWRITEABLE, set IORESULT to INOTWRITEABLE.
- If not FLOCKED, set IORESULT to IFILEUNLOCKED.
- The AM may set IORESULT.

fwritepaoc

Pascal Declaration

```
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 (maximum 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

pointer to t	← sp+16
pointer to a	← sp+12
aleng	← sp+8
rleng	← sp+6
return address	← sp+4
	← stack pointer

Action

1. 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.
2. Call FWRITEBYTES to write ALENG bytes from the packed array of characters to the file.

Errors

- FWRITECHAR may set IORESULT.
- FWRITEBYTES may set IORESULT.

fwritereal

Pascal Declaration

```
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 (maximum 255).
D	The number of digits after the decimal point.

Stack

Stack contents	Offset
	← sp+16
pointer to t	← sp+12
pointer to x	← sp+8
w	← sp+6
d	← sp+4
return address	← stack pointer

Action

1. Call STRWRITE (compiler-generated call to FWRITESTRREAL) to write the real to a string.
2. If IORESULT is INOERROR, call FWRITEBYTES to write the string to the text file.

Errors

- FWRITESTRREAL may set IORESULT.
- FWRITEBYTES may set IORESULT.

fwritestr

Pascal Declaration

```
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 (maximum length of 80).
RLENG	The field width to be written into. Note that this must be no more than 255 + strlen(S).

Stack

	← sp+14
pointer to t	← sp+10
pointer to s	← sp+6
rleng	← sp+4
return address	← stack pointer

Action

1. If RLENG < 0, then set RLENG to the length of S.
2. If RLENG > the length of S, call FWRITECHAR to write a space in a field width of RLENG - strlen(S) (i.e., write that many spaces) and set RLENG to the length of S.
3. Call FWRITEBYTES to write the string to the file.

Errors

- FWRITECHAR may set IORESULT.
- FWRITEBYTES may set IORESULT.

fwritestrbool

Pascal Declaration

```
procedure fwritestrbool(var s:      string;
                        var p2:     integer;
                            b:      boolean;
                            rleng:  shortint);
```

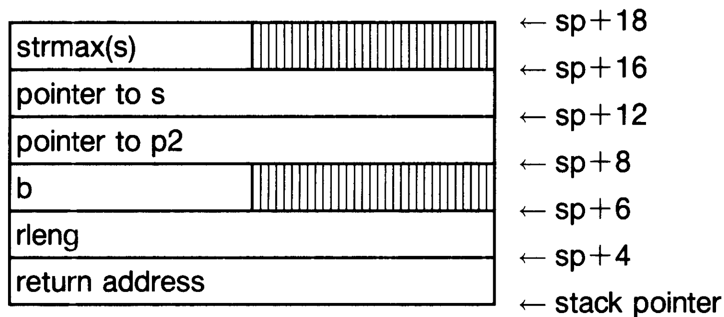
Purpose

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 (maximum 255).

Stack



Action

1. 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.

fwritestrchar

Pascal Declaration

```
procedure fwritestrchar(var s:      string;
                        var p2:     integer;
                            ch:     char;
                            rleng:  shortint);
```

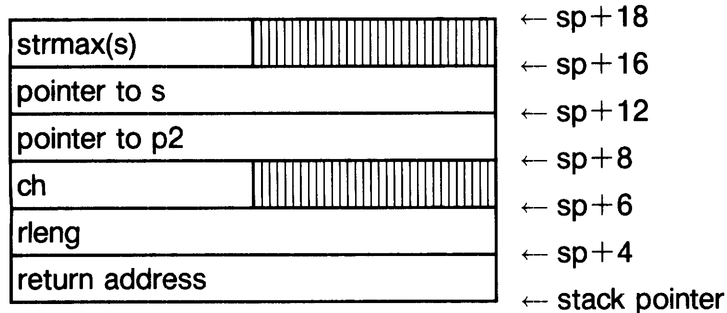
Purpose

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



Action

1. Convert CH to a string of length 1.
2. If RLENG < 1, then set RLENG to 1.
3. Call FWRITESTRSTR with S, P2, the constructed string, and RLENG.

Errors

- FWRITESTRSTR may set IORESULT.

fwritestrenum

Pascal Declaration

```
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 (maximum 255).
P	The compiler-generated table of string values for the enumerated type.

Stack

Stack contents	Offset
	← sp+22
strmax(s)	← sp+20
pointer to s	← sp+16
pointer to p2	← sp+12
i	← sp+10
rleng	← sp+8
p	← sp+4
return address	← stack pointer

Action

1. Extract the identifier in table P indexed by I.
2. 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.

fwritestrint

Pascal Declaration

```
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.
RLENG	The field width to be written into.

Stack

strmax(t) ▥	← sp+20
pointer to t	← sp+18
pointer to p2	← sp+14
i	← sp+10
rleng	← sp+6
return address	← sp+4
	← stack pointer

Action

This routine is written in assembly for speed.

1. Remember sign of I.
2. 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.
3. If RLENG > the length of the dummy string (+1 if the sign is negative), put RLENG − the length of the dummy string (−1 if the sign is negative) in S starting at S[P2].
4. If sign is negative, put a '-' after the spaces (if any).
5. After the sign, if any, put the dummy string.
6. If the length of S has changed, update S[0].
7. Add the number of characters written to S to P2.

Errors

- If (P2 < 1) or (P2 > strlen(S)+1), set IORESULT to ISTROVFL.
- If the write would extend the length of S past strmax(S), set IORESULT to ISTROVFL.

fwritestrpaoc

Pascal Declaration

```
procedure fwritestrpaoc(var s:      string;
                        var p2:     integer;
                        var a:      window;
                            aleng,
                            rleng:  shortint);
```

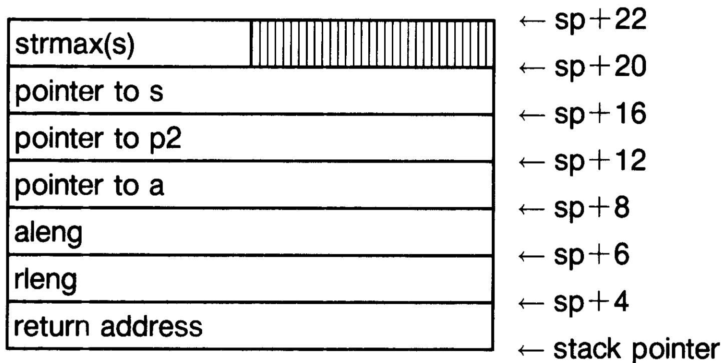
Purpose

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 (maximum 255).
RLENG	The field width to be written into.

Stack



Action

1. Convert A into a string of length ALENG.
2. Call FWRITESTRSTR with S, P2, the constructed string, and RLENG.

Errors

- Boundary errors may arise if ALENG > 255.
- FWRITESTRSTR may set IORESULT.

fwritestrreal

Pascal Declaration

```
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

Stack	Offset
strmax(r)	← sp+22
pointer to r	← sp+20
pointer to p2	← sp+16
pointer to x	← sp+12
w	← sp+8
d	← sp+6
return address	← sp+4
	← stack pointer

Action

1. 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.
2. If it will fit, move this string representation into R starting at P2 and update the length of R if necessary.
3. Set P2 to one past the character written.

Errors

- If (P2 < 1) or (P2 > strlen(S)+1), set IORESULT to ISTROVFL.
- If the write would extend the length of S past strmax(S), set IORESULT to ISTROVFL.

fwritestrstr

Pascal Declaration

```
procedure fwritestrstr(   var s:       string;
                          var p2:      integer;
                       anyvar t:       string255;
                              rleng:   shortint);
```

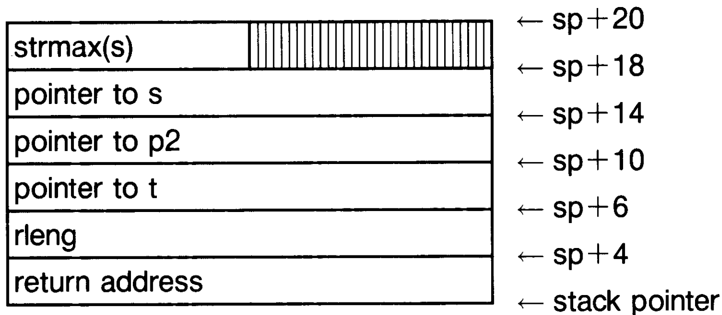
Purpose

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



Action

1. If RLENG < 0, then set RLENG to the length of T.
2. If RLENG > the length of T, replace S[P2] to S[P2+RLENG-(strlen(T))-1] with spaces and add RLENG-(strlen(T)) to P2.
3. Copy RLENG characters of T into S starting at S[P2].
4. Add RLENG to P2.
5. If P2+strlen(T)-1 > strlen(S), set length of S to P2+strlen(T)-1.

Errors

- If (P2 < 1) or (P2 > strlen(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.

fwritestrword

Pascal Declaration

```
procedure fwritestrword(var s:    string;
                        var p2:   integer;
                            i,
                            rleng: shortint);
```

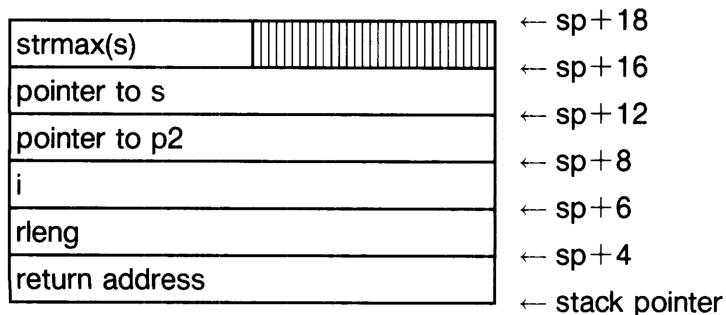
Purpose

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



Action

1. Call FWRITESTRINT with S, P2, I, and RLENG. Note that I is passed by value so this is possible.

Errors

See FWRITESTRINT.

fwriteword

Pascal Declaration

```
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 (maximum 255).

Stack

Stack contents	Offset
	← sp+12
pointer to t	← sp+8
i	← sp+6
rleng	← sp+4
return address	← stack pointer

Action

1. Call FWRITEINT with T, I, and RLENG. Note that I is passed by reference so this is possible.

Errors

See FWRITEINT.

scantitle

Pascal Declaration

```
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 file kind 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

Stack	Offset
func. result	← sp+26
pointer to fname	← sp+24
pointer to fvid	← sp+20
pointer to ftitle	← sp+16
pointer to fsegs	← sp+12
pointer to fkind	← sp+8
return address	← sp+4
	← stack pointer

Action

1. Call `ZAPSPACES` with `FNAME`.
2. 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.
3. Extract the volume id.
4. If there is an illegal volume id then return `FALSE`. (Legal volume ids include `'#`*nn*`'`, `'#`*nn*`:'`, `':'`, `'*'`, `'*:'`, and `'cccccccccccccccc:'`.)
5. The colon (if any) is removed.

6. If after removing the colon '*' is left, use the system volume id.
7. If nothing is left, use the default (prefix) volume id.
8. Set FVID to the volume id.
9. Extract the file size specifier.
10. If no legal size specifier is found, set FSEGS to 0. (Legal size specifiers include [*] and [*non-negative integer*]).
11. If it is [*] then set FSEGS to −1. Otherwise, set FSEGS to the non-negative integer.
12. Set FTITLE to "all the rest" (i.e., the file id without spaces and control characters, volume identifier, colon, and file size specifier.)
13. Call SUFFIX with FTITLE and set FKIND to the file kind returned.

Errors

- No file name left after call to ZAPSPACES, return FALSE.
- Illegal volume id, return FALSE.

suffix

Pascal Declaration

```
function suffix(var ftitle: string): filekind;
```

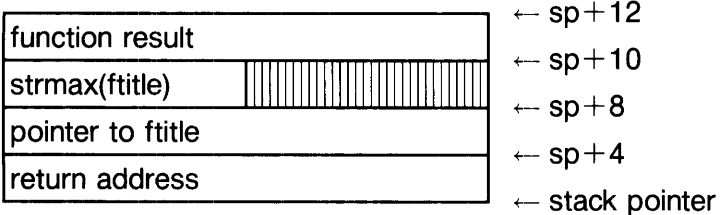
Purpose

To determine the file kind associated with the suffix of a filename.

Parameters

FTITLE   The filename.

Stack



Action

1. Search suffixtable from UNTYPEDFILE to LASTFKIND.
2. If a suffix in the table matches suffix of ftitle (uppercased) then return the FILEKIND index.
3. Stop when first match is found.
4. If no match is found, return SUFFIX:=DATAFILE.

Errors

None.

zapspaces

Pascal Declaration

```
procedure zapspaces(var s: string);
```

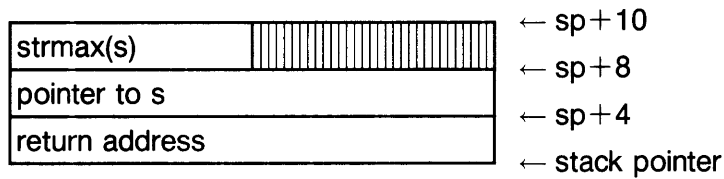
Purpose

To remove all spaces and control characters from a string.

Parameters

s Any string.

Stack



Action

1. String is scanned and all characters <= `chr(32)` (ASCII blank) or = `chr(127)` (ASCII del) are removed.

Errors

None.

Directory Access Methods 5

Reference Specification for Directory Access Methods (DAMs)

This section describes what a Directory Access Method (DAM) must be able to do in order to 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 (FIB) for any type of file. The second is a unit number (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.

Note: 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 (unless no access is necessary) should 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 null 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 I/O subsystem in unforeseen ways.

Calling DAMs

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 that FRECSIZE is greater than zero.
- 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 equal to 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, pathname , 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 (e.g., non-contiguous file systems), FPEOF should be set to MAXINT.

AM — 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^[UNTYPEDFI LE]
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 chooses.

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 DAMs 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 file 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 DAMs 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(ILOSTFILE).

If the volume is right but the file name can't be found (e.g., if someone else purged it), set IORESULT to ord(ILOSTFILE). 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(ILOSTFILE).

If the volume is right but the file name can't be found (e.g., if someone else purged it), set IORESULT to ord(ILOSTFILE). 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.

Note: 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 is set to 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 null string[1] '' should be returned.

GETVOLUMEDATE and SETVOLUMEDATE

The actual parameter F is a variable of type DATETIMEREC instead of a 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 is zero, the date is invalid; some file systems may use a YEAR value of 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", i.e., it doesn't report on its progress to the CRT.

On entry, F is a 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.

---

[1] The "null string" is a string whose length is zero.

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 (e.g., 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 a 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[2] 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 IORESULTs 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 CATENTRYs 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).

[2] Some of the fields of CATENTRY may not make sense for certain kinds of mass storage devices. For example, CSTART is not useful for SRM.

- 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 a 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 a 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 and later LIF DAMs are supersets of the LIF specification, allowing access to Series 200 BASIC files and providing compatibility with Series 200 Pascal release 1.0 file naming conventions. The extensions are only in the allowable names for files—Pascal Workstation 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 installifdam;
implement
   .
   .
   .
end;  {lifmodule}

begin  {program instlifdam}
  installifdam;
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 INSTALLIFDAM, 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 is 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:        fib;              {file descriptor}
                        unum:     unitnum;          {logical unit number}
                        request:  damrequesttype);  {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) and (escapecode<>-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 MISC, 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.

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 Workstation LIF 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 MISC; 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 lowercase letters are distinct. File names stored in LIF directories are always exactly ten 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 operation 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 ten 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 ten 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, e.g., "A" for ".ASC".
4. If the name is now less than ten characters long, it is extended by appending underscores "_" to ten characters.

Using this algorithm, we would have the following examples:

Typed:	New File Name:
"A.ASC"	"AA________"
"charlie"	"charlie "
"123456789.TEXT"	"123456789T"
"KBDSUPPORT"	Rejected because it would be confused with the transformation of "KBDSUPPOR.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 a 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 DOOPENDIRECTORY

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 `CATENTRY`s.

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 the largest available block of free space.

SPACE<0 — Request for "about half the free space"—half the largest free hole, or the second largest free hole, whichever is the greater.

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 $512 \times nnn$ bytes. The use of 512-byte blocks rather than bytes or sectors 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 two sectors long. Another field, DSIZE, tells the number of sectors allocated for the directory. Each sector holds up to eight directory entries.

Files with FTYPE equal to zero are purged files. The last directory entry has FTYPE equal to -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 equal to zero is not a reliable way to measure open space; the file may have been broken up and part of it given away. You must check other directory entries to find the size of the hole.

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 far 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` or `SAVE`.

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 FIBs; 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 a nonzero IORESULT.

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             determined 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            determined 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
feft
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            determined 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 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 null 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 zero and FPEOF to the number of entries in the caller's array. After the first call, set FPOS:=FPOS+FPEOF to catalog consecutive files.

```
FIB setup
  funit
  fvid
  fwindow     points to an array of CATENTRYs
  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.

File Operations 6

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 3.0 file support system, particularly certain service requests to the Directory Access Methods (DAMs).

The source code for FILEPACK is in file FILEPACK.TEXT on the EXAMPL: disc. As with all material found on EXAMPL:, it not supportable by Hewlett-Packard, either in the field or in the factory.

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 DAMs 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(filenumber: 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;
        passarayptr = ^passarray;

    var
        catfib, passfib: ^fib;
        catentptr       : ^catarray;
        wordlist, optionlist: passarayptr;
```

Function MIN

Function MIN returns the lesser of two integers.

```
  function min(a, b: integer): integer;
   begin
    if a<b then
     min := a
    else
     min := b
   end; { min }
```

Function 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.

```
  function ioerrmsg(var msg: string): boolean;
   begin
    if ioresult=ord(inoerror) then
     ioerrmsg := false
    else
     begin
      ioerrmsg := true;
      getioerrmsg(msg, ioresult);
     end;
   end; { ioerrmsg }
```

Procedure IOCHECK

Procedure IOCHECK tests to see if IORESULT is non-zero (not equal to ord(INOERROR)) and if so, escapes with ESCAPECODE equal to −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 iocheck;
   begin
    if ioresult <> ord(inoerror) then
     escape(-10);
   end; { iocheck }
```

Procedure BADIO

Procedure BADIO sets IORESULT to the specified value and escapes.

```
procedure badio(iocode: iorsltwd);
 begin
  ioresult := ord(iocode);
  escape(-10);
 end; { badio }
```

Function 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 unitnumber(var fvid: vid): boolean;
 var scanning: boolean;
     i: shortint;
 begin
  unitnumber := false;
  zapspaces(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 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.

```
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 }
```

Procedures ANYTOMEM and MEMTOANY

These two procedures provide the mechanism by which files (primarily `TEXT`) are translated. In the Pascal 3.0 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 (AMs) 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 AMs 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 one) 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 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 anytomem(ffib: fibp; anyvar buffer: bigptr; maxbuf: integer);
 var bufrec    : ^string255;
     bufptr    : ^char;
     leftinbuf: integer;
 begin { anytomem }
  bufptr     := addr(buffer^);
  bufptr^    := chr(0);   { data coming }
  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 result form 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} 2;  {RQ/SFB 3/15/84  3.0 BUG}
        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;
         LEFTINBUF := LEFTINBUF -1;  {RQ/SFB 3/15/84  3.0 BUG}
         if ioresult = ord(ieof) then
          bufptr := addr(bufptr^, 1);
        end;  { end of line }
       if ioresult = ord(ieof) then
        begin  { end of file }
         bufptr^  := chr(2);
         ioresult := ord(inoerror);
         feof     := true;
        end;  { end of file }
       iocheck;  { check ioresult from readbytes }
      end;
     if not((leftinbuf < 259) or feof) then
      begin  { set up 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 MEMTOANY

Procedure MEMTOANY writes the contents of the buffer into the destination file.

```
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, writeeol, bufptr^, bytes, fpos);
          bytes := 0;
         end;
      2: begin  { end file }
          call(am, ffib, flush, bufptr^, bytes, fpos);
          bytes := -1;
         end;
      3: bytes := -1;  { end of buffer }
      otherwise ioresult := ord(ibadrequest);
     end;  { case }
     iocheck;
    until bytes < 0;
   end;
 end;  { memtoany }
```

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 DAMs (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, and
   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.
FOPTSTRING	A pointer to the optional third parameter in open statements, a string which may contain, e.g., 'SHARED' or 'EXCLUSIVE'. May be initialized to the null string 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 and end-of-line marker has been read.
FREPTCNT	Temporary field used by AMs, must initially be 0.
FLASTPOS	Temporary field used by AMs, must initially be −1.
FBUFCHANGED	Temporary field used by AMs, must initially be FALSE.
FNOSRMTEMP	Temporary field used by SRM, must initially be TRUE.

```
procedure setupfibforfile(var filename      : fid;
                          var lfib          : fib;
                          requireddirectory : 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 requireddirectory)) then
      begin
       iocheck;
       if unitnumber(fvid) then
        badio(znodevice);
      end;
     fkind        := lkind;
     feft         := efttable^[lkind];
     fpos         := segs * 512;
     freptcnt     := 0;
     flastpos     := -1;
     fanonymous   := false;
     fmodified    := false;
     fbufchanged  := false;
     fstartaddress:= 0;
     pathid       := -1;
     foptstring   := nil;
     fnosrmtemp   := true;
     flocked      := true;
     feof         := false;
     feoln        := false;
    end
   else
    badio(ibadtitle);
  end;  { setupfibforfile }
```

Procedure 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 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 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 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 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 FIBs 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, STARTADDRESS 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 OUTSIZE 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 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    := efttable^[fkind];
       outsize := 0;
      end
     else
      begin
       fkind   := outfkind;
       feft    := outeft;
       if fpos = 0 then  { no size was specified }
        fpos   := outsize;
      end;
     fstartaddress := outfstarta;
```

```
 if strlen(ftitle) = 0 then
  begin  { x -> volume }
   call(unitable^[funit].dam, outfib, funit, openvolume);
   iocheck;
   if fpeof < outsize then
    badio(inoroom);
  end  { x -> volume }
 else
  begin  { x -> file }
   if writeover then
    overcreate := overwritefile
   else
    overcreate := createfile;
   call(unitable^[funit].dam, outfib, funit, overcreate);
   iocheck;
   if fpeof < outsize then
    begin  { try to stretch file }
     fpos := outsize;
     call(unitable^[funit].dam, outfib, funit, stretchit);
     iocheck;
     if outsize > fpeof then
      badio(inoroom);
    end;
  end;  { x -> file }
 fpos      := 0;
 fwriteable:= true;
end;  { with outfib }

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
 outsize := -1;

repeat  { move the file }
 with infib do
  if format then
   begin  { formatted filecopy }
    anytomem(addr(infib), buf, bufsize);
    if feof then
     outsize := 0;
   end
  else
   begin  { unformatted filecopy }
    if bufsize > outsize then
     movesize := outsize
    else
     movesize := bufsize;
    call(unitable^[funit].tm, addr(infib), readbytes, buf, movesize,
         fpos);
    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;
         outsize := outsize - movesize;
        end;
    until outsize  = 0;

    release(lheap);
    closeinfile(infib);
    closeoutfile(outfib, keepit);

  recover
   begin
    release(lheap);
    saveio    := ioresult;
    saveesc   := escapecode;
    closeinfile(infib);
    closeoutfile(outfib, purgeit);
    escape(saveesc);
   end;
  end;  { filecopy }
```

Procedure VOLUMES

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 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;  { volumes }
```

Procedure REPACK

Procedure REPACK calls on the DAM using the CRUNCH request to repack the indicated volume.

```
 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 }
```

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 a couple of 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 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);
       iocheck;
       freadable := true;
      end;
    end;
  end;  { opendir }
```

Procedure 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 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 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, and
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.
- CEXTRA1 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 UEOVBYTES (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 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 set up default values }
      setstrlen(cname, 0);      { volume name }
      cpsize  := maxint;        { size in bytes }
      cextra1 := 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        { -1 retains old values }
     cextra1 := entries;        {  0 selects default
    if bytes > 0 then
     cpsize := bytes;           { -1 retains old value }
    if cpsize = 0 then
     badio(ibadvalue);
    zapspaces(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 MAKEDIR

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 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;  { makedir }
```

Procedure MAKEFILE

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 makefile(filename : fid);
 var outfib : fib;
 begin
  with outfib do
   begin
    setupfibforfile(filename, outfib, true);
    call(unitable^[funit].dam, outfib, funit, createfile);
    iocheck;
    fwriteable := true;
    fleof      := fpeof;  { cause file size to be retained }
    closeoutfile(outfib, keepit);
    iocheck;
   end;  { with }
 end;  { makefile }
```

Procedure ENDCAT

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 endcat;
 begin
  if catfib <> nil then
   begin
    closedir(catfib^);
    release(catfib);
    catfib := nil;
   end;
 end;  { endcat }
```

Procedure STARTCAT

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); and 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 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  { startcat }
  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;  { startcat }
```

Procedure CAT

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, usually an HP standard type code.
BLOCKSIZE	The size of a logical block in bytes, 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 cat(filenumber :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);
 begin
```

```
if catfib = nil then escape(-3);
with catfib^, unitable^[funit] do
 begin
  if not freadable then
   badio(inotopen);
  if (filenumber >= 0) and
   ((filenumber < fpos) or
   ((filenumber >= fpos + catlimit) and (fpeof = catlimit))) then
    begin
     fpos  := filenumber;
     fpeof := catlimit;
     call(dam, catfib^, funit, catalog);
     iocheck;
    end;
  if (filenumber < fpos) or (filenumber >= fpos + fpeof) then
   filename := ''
  else
   with catentptr^[filenumber - 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;  { cat }
```

Procedure 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 FIBs 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 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  := ioresult;
     saveesc := escapecode;
     closedir(infib);
     closedir(outfib);
     ioresult := saveio;
     if saveesc <> 0 then
      escape(saveesc);
    end;
 end;  { duplicate }
```

Procedure REMOVE

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 remove(filename: fid);
 var infib : fib;
 begin
  setupfibforfile(filename, infib, true);
  with infib do
   call(unitable^[funit].dam, infib, funit, purgename);
  iocheck;
 end;  { remove }
```

Procedure 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 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(ibadtitle);
  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 ENDPASS

Procedure ENDPASS releases the heap memory which was allocated by procedure STARTLISTPASS in order to manipulate passwords. Procedure ENDPASS should always be called after listing and changing passwords is complete.

```
procedure endpass;
 begin
  if passfib <> nil then
   release(passfib);
 end;  { endpass }
```

Procedure STARTLISTPASS

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 PASSENTRY records.
3. Initialize the FIB:
    a. FWINDOW is a pointer to WORDLIST^,
    b. FPOS indicates which password should be listed first (initially 0), and
    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 PASSENTRY 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 PASSENTRY is a bit pattern which is used to match these passwords with their attributes.

```
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;  { startlistpass }
```

Procedure LISTATTRIBUTE

Procedure LISTATTRIBUTE searches the list of legal password attributes produced by procedure STARTLISTPASS and returns the name of the one which is the *n*th 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 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;  { listattribute }
```

Procedure LISTPASSWORD

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 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) = p 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;  { listpassword }
```

Procedure CHANGEPASSWORD

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 PASSENTRY record.
3. Set these fields in the FIB:
   a. FWINDOW is a pointer to the new PASSENTRY record,
   b. FPOS is 0, and
   c. FPEOF is 1.
4. Call the DAM using the SETPASSWORDS request.

```
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;
  zapspaces(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, 1, 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 }
  zapspaces(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;  { changepassword }
```

Procedure PREFIX

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.

```
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);
  iocheck;
 end;  { prefix }


end.  { module filepack }
```

CPU Interrupt Handling 7

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 I/O library procedures. That is covered in the *System Devices* chapter of the *Pascal 3.0 Procedure Library* manual. Also, the system designers took pains to provide an easy-to-use special case for intercepting keyboard processor interrupts, which is discussed in the *Keyboard* chapter.

Interrupts are a special case of exception processing. The 68xxx 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. Note: do not confuse `RESET` with the `RESET` key on the keyboard.

In the Series 200 machines, 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 HP-IB port; also DMA card and HP 98620A.
Level 7	Powerfail interrupt, if present; also `RESET` or `Shift`-`Break` (or `SHIFT`-`PAUSE` or `SHIFT`-`PSE`) and `Ctrl`-`Shift`-`Break` (or `CTRL`-`SHIFT`-`PAUSE` or `CTRL`-`SHIFT`-`PSE`).

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 stack "SSP".

An interface requests interrupt service over three signals (`IPL0'`, `IPL1'`, and `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 current CPU priority 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. The latter is not supported on Series 200; only "autovectors" in boot ROM to high memory are supported. 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 Series 200 machines, but as of this writing, only the first method is actually used by HP interfaces.

In the Series 200 machines, all 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 leading to the service routines in these RAM areas. The boot ROM routine puts initial values in those locations 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.

When the system is booted, it begins executing 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 I/O 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 I/O subsystem is to deal with interrupts at that interrupt level. 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 = {isr information block}
     packed record
       intregaddr: charptr;     (interrupt register address)
       intregmask: byte;        (interrupt register mask)
       intregvalue: byte;       (interrupt register target value after masking)
       chainflag: boolean;      (chaining flag)
       proc: isrproctype;       (isr)
       link: pisrib;            (pointer to next isrib in linked list)
     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;
```

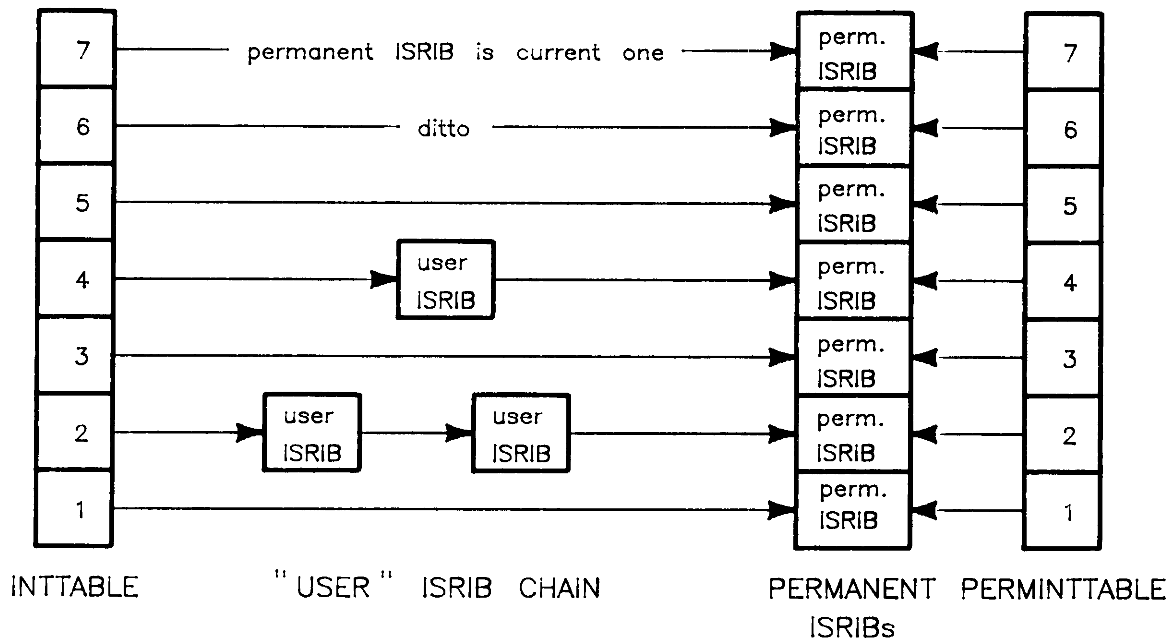
Every ISR is a procedure which takes, as its single parameter, a pointer to an ISR information block (ISRIB).

The ISRIB gives all the information needed to test whether any single interface is interrupting. INTREGADDR is a pointer into the I/O portion of address space; it accesses a one-byte register indicating whether the interface is requesting interrupt service. This byte is *and*ed 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 I/O select code is allocated 65 536 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, that is, the second 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 I/O 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 ISRs are the ones the system expects to be present. They are *not* removed automatically whenever a program terminates, whereas user ISRs are removed then. User ISRs are simply temporary ones stacked at the front of the list by `ISRLINK` or removed by `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* want 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 as to clear the interrupt cause/request.

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 unserviced interface will usually 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 automatically clear interrupt request when their data is read; others must be cleared 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 HP-IB 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 action.

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 clears 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 68xxx 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 provided 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.

NOTE

This is different from writing ISRs to work as a part of the Pascal device I/O library. The procedure to be described now bypasses the I/O 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 (i.e., 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 one 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 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 for you when your program ends. 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 **$STACKCHECK$**, **$RANGE$**, **$OVFLCHECK$**, and **$DEBUG$**. 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 I/O: reading a character, reading the clock, etc. Unfortunately, the keyboard usually communicates with the 68xxx 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 in the first ISR.

There is potential for this problem to appear elsewhere, especially in HP-IB service since the TI9914 chip often communicates by interrupt.

The Keyboards 8

Introduction

This chapter introduces the keyboard, a surprisingly complicated subject. The keyboard hardware includes its own microprocessor which not only handles keystrokes but also manages the "knob" (if your keyboard has one) and maintains several timers capable of interrupting the CPU. There are at present three main types of keyboards:

98203A	71 keys
98203B	105 keys
46020A	107 keys

This chapter will discuss all of the above keyboards. The 98203A is physically the smallest of the keyboards; the 98203B is larger, and can be attached or detached from the rest of the computer. The 46020A keyboard is an HP-HIL (Hewlett-Packard Human Interface Loop) device, but, except for the hardware implementation, its functionality and method of communicating with the 68xxx are basically the same. One exception is the absence of a knob on the 46020A.

These various keyboards are distinguished by the configuration jumpers on the keyboards themselves. At present, the 98203A and 46020A keyboards are detached from the mainframe while the 98203B keyboard can be either attached or detached. The 46020A keyboard differs from the other two in the following way:

- The HP-HIL keyboard is just an HP-HIL device which happens to be a keyboard; there are many other kinds of HP-HIL devices which talk the same language and use the same interfaces. In other words, this generalized interface can talk to many devices, one of which is a keyboard.
- The 98203 keyboards are just keyboards; there are no other devices which are connected to the computer in the same way. It is a very specific interface that can be used for nothing else.

Associated 68xxx 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.

The 46020A HP-HIL keyboard, Hewlett-Packard's recently-introduced keyboard, is the "new kid on the block." It is a new standard keyboard, and from the casual user's point of view, operates virtually identically to the previous keyboards. In fact, the interface to the timers, real-time clock and beeper has not changed at all, other than the enhancements. The interface to the keyboard is also nearly identical to the old definition with the only exceptions being changes in some of the keycodes returned. These changes are a result of the new 46020A keyboard layout.

Summary of Keyboard Controller 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 (four voices with the HP-HIL keyboard controller).

Keyboard:

- Keycodes: 98203A keyboard—71 keys on keyboard. 98203B keyboard—105 key matrix positions, qualified by `SHIFT` and `CTRL` keys. 46020A keyboard—107 key matrix positions qualified by `Shift`, `Ctrl` and `Extend Char` keys.
- Scanning: 2-key rollover with trailing-edge debounce (not available on the 98203A keyboard). N-key rollover with 46020A keyboard.

- Interrupt 68xxx on keystroke.
- Auto-repeat depressed key.
- Set auto-repeat rate.
- Set auto-repeat delay before first repetition.

HP-HIL Port (46020A only)

- Input capability—autopolling.
- Generalized HP-HIL I/O.
- Keyboard "cooking."

Rotary Pulse Generator (RPG or "Knob;" 98203A/B only)

- Detect rotation direction; 120 pulses per 360 degrees.
- Accumulate pulse count during specified period.
- Set up knob interrupts.
- Read knob pulse count.

Keyboard "Cooking" and "Raw" Data

HP-HIL 46020A keyboards return both up-stroke and down-stroke keycodes for up to 128 keys. To minimize changes in the operating systems and boot ROM, the keyboard controller has the capability of mapping the HP-HIL keycodes into the keycodes and protocol expected by the series 200 operating systems. At powerup, the default is to map all keyboards on the loop through this protocol conversion algorithm. This protocol conversion (keyboard "cooking") can be selectively or collectively enabled or disabled for any or all keyboards on the loop. That is, any keyboard can be specified as "cooked" or "uncooked." The functions provided by the protocol conversion are:

- Map the 46020A down-stroke keycodes into series 200 standard keycodes.
- Block all 46020A up-stroke keycodes (except for a few special keys).
- Maintain the status of the `Shift` and `Ctrl` keys and return the state of these keys with the mapped keycodes.
- Implement the auto-repeat function with programmable delay and repeat rate.
- Implement the level 7 interrupt reset function (`Shift`-`Break`).

HP-HIL keyboards can have their data handled by the 8042 in either "raw" mode or "cooked" mode. Raw mode (enabled by clearing bits in the 8042 register `KBDSADR`, which is at location `$79`) causes all keystrokes to be interpreted as generic HP-HIL data—they interrupt through `STATUS5HOOK` and `STATUS6HOOK`. Cooked mode causes keystrokes to be understood as such, and thus they interrupt through the regular `KBDISRHOOK` channel. In cooked mode, if there are multiple keyboards on the HP-HIL loop, there is no way to distinguish which keyboard a keystroke came from. If an application requires the separation of data from keyboards at different HP-HIL addresses, the keyboards must be enabled for raw mode (HP-HIL protocol) data transfers which include the addressing information.

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 `Break` (or `PAUSE` or `PSE`) and then typing on the keyboard. Hold down `Ctrl` and press `Clear Line` (or `CLR LN` or `CLR L`) to clear the typeahead buffer. Press the "Continue" key to allow the system to free-run again.

The keyboard, like all HP Pascal text files, exhibits "lazy" I/O 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 `Return` (or `ENTER`) key or the limit of the string's length, whichever happens first. You *cannot* backspace back into a valid part of the string from beyond the string's limit. 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 `Return` or `ENTER` key). Note that the carriage return is converted into a space (`chr(32)`). 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. Note: `EOF` is undefined for `INPUT` and `KEYBOARD`[1] and is never returned by these standard files.

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 Back Space)
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.

[1] The standard file `KEYBOARD` works exactly as the standard file `INPUT`, except that keystrokes are not echoed to the screen. See *Non-Echoing Read*, next.

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 ReadNoEcho1(keyboard, output);
var
  Character:            char;
  keyboard:             text;
begin
repeat
  read(keyboard, Character);
  write(output,'You pressed ');
  if Character>=chr(32) then                        {printable character}
    write(output,'"',Character,'", which is ');
  writeln(output,'CHR(',ord(Character):1,').');
until false;                             {press [STOP] to end the program}
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 ReadNoEcho2(output);
var
  Character:            char;
  MyFile:               text;
begin
reset(MyFile,'#2:');                          {open to logical unit two}
repeat
  read(MyFile, Character);
  write(output,'You pressed ');
  if Character>=chr(32) then                        {printable character}
    write(output,'"',Character,'", which is ');
  writeln(output,'CHR(',ord(Character):1,').');
until false;                             {press [STOP] to end the program}
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 physically resides in the 98203B keyboard, and in the mainframe for the other two keyboards, it can be triggered by writing an ASCII bell character to the standard file OUTPUT:

```
write(output,#G);      {ASCII "bell" is control-G, or CHR(7)}
```

To control tone and duration, it is necessary to 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 module SYSDEVS, which is 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. Thus, Pascal programs doing READs 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 AvoidHangs1(keyboard, output);
import uio;
var
  keyboard:             text;
  Character:            char;
  I:                    integer;
begin
writeln(output,'Press any key.  Hurry!');
for I:=1 to 3000 do
  begin
  if not unitbusy(2) then             {"busy" means typeahead buffer empty}
    begin
    writeln(output,'You beat me!');
    read(keyboard,Character);           {eat character in typeahead buffer}
    halt;                             {stop here; don't print other message}
    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 SYSDEVS is exported an integer function SYSCLOCK, which returns the number of centiseconds (hundredths of a second) since midnight.

```
program AvoidHangs2(keyboard, output);
import uio,sysdevs;
const
  TimeLimit=           200;                 {200 centiseconds = 2 seconds}
var
  keyboard:            text;
  Character:           char;
  Start:               integer;
  GotOne:              boolean;
begin
writeln(output,'Press any key within two seconds!');
Start:=sysclock;
GotOne:=false;
repeat
  if not unitbusy(2) then            {"busy" means typeahead buffer empty}
    begin
    writeln(output,'You beat me!');
    read(keyboard, Character);  {eat the character in the typeahead buffer}
    GotOne:=true;
    end;
until GotOne or (sysclock>(Start+TimeLimit));
if not GotOne then
  writeln(output,#G,'Wow, you ARE slow!');
end.
```

Using the System Clock and Calender

You 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;
```

SYSDEVS exports these things:

```
procedure sysdate(var thedate: daterec);
procedure setsysdate(thedate: 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.

NOTE

Even though SYSDEVS exports the procedures, the module CLOCK must be in INITLIB (or executed from the command interpreter) to get the functionality.

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.

NOTE

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 the 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 on the keyboard except [Shift], [Ctrl], [Extend char] and [Reset] ([Shift]-[Break] or [SHIFT]-[PAUSE] or [SHIFT]-[PSE]). [Shift], [Ctrl] or [Extend char] are separated from all the other keys to allow for the computer to detect simultaneous depression of the [Shift], [Ctrl], or [Extend char] 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 (NMI) 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 68xxx communicate. When the keyboard has something to say, it usually interrupts the 68xxx on priority level one. Under certain circumstances it can force NMI (Non-Maskable Interrupt), but not for keystrokes. Conversely, the 68xxx interrupts the keyboard processor when it wants to issue a command. The interrupts initiate or terminate communication; data is actually exchanged via memory-mapped I/O, meaning certain locations in the 68xxx'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 68xxx 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, an ISR called A804XISR (located in module A804XDVR) executes and gets the Data and Status bytes from the memory-mapped 804x registers at addresses $428001 and $428003, respectively. Then, A804XDVR notes the most significant four bits of the Status byte, which determines the source of the interrupt. The upper four bits have the following interpretation:

0 (0000):	Unimplemented. Do nothing.
1 (0001) through 3 (0011):	Either the 10-msec Periodic System Interrupt (PSI), indicated by bit 4 of the Status byte, one of the special timers, indicated by bit 5, or both. The "special" timers are cyclic, delay, and match. Transfers control to TIMERISRHOOK.
4 (0100):	Data request other than through HP-HIL. The data is in the data register.
5 (0101):	HP-HIL status change; calls STATUS5HOOK. The HP-HIL status code is in the data register.
6 (0110):	HP-HIL data available; calls STATUS6HOOK. The data is in the data register.
7 (0111):	Unimplemented. Do nothing.
8 (1000) through 11 (1011):	Keypress on cooked keyboard; the data is in the data register. Call KBDTRANSHOOK. If DOIT comes back still true, then call KBDISRHOOK also. Bit 5 of the Status byte indicates "not Ctrl," and bit 4 of the Status byte indicates "not Shift."
12 (1100) through 15 (1111):	Knob rotation; the pulse count is in the data register. Call RPGISRHOOK. Bits 7–6 are 11; bit 5 of the Status byte indicates "not Ctrl," and bit 4 of the Status byte indicates "not Shift."

Since the A804XISR calls the "hook" routines, you can plug in your own routines to which to transfer control. Then, if you wish, you can call the ISR which would have been called had your ISR not intercepted the interrupt.

When a keyboard interrupt is detected, A804XISR reads Status and Data. If Status indicates a keystroke, the KBDTRANSHOOK routine executes. The standard KBDTRANSHOOK performs two main tasks, both for the 46020A keyboards only:

- It keeps track of the pressed/release states of the Extend Char keys; and,
- If the keyboard is in "system mode," it translates softkeys to their system definition keycodes (e.g., the **Step** softkey, etc.).

Once KBDTRANSHOOK has executed, KBDISRHOOK is called. KBDTRANSHOOK can have left the variable DOKEY either TRUE or FALSE; KBDISRHOOK should only "do" anything if DOKEY is TRUE at entry. The standard KBDISRHOOK routine must translate the keycode into a character, taking into account such factors as the language of the keyboard (Katakana, French, Spanish...), the type of keyboard (98203A/B or 46020A), whether the Shift key was down or the Caps (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-Back Space, 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, at some time, 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.

Note that if you take over KBDISRHOOK without taking over KBDTRANSHOOK, softkeys from the 46020A will not return the codes shown in the keyboard diagram if the keyboard is in system mode (KBDSYSMODE = TRUE). To get the codes shown while in system mode, you must prevent KBDTRANSHOOK from converting the keyboard data to system definition keycodes.

Knob interrupts are handled similarly, since the operating system 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 wanted it to.

In summary: the keyboard processor sends Status and Data to the 68xxx for each keypress. On the 68xxx side, the ISR called A804XISR (in module A804XDVR) decodes Status and Data and calls a hook which may derive an ASCII character to be 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 some procedure variable hooks exported from SYSDEVS. An exactly analogous capability is provided for intercepting timer interrupts.

```
type
   kbdhooktype=      procedure(var statbyte,databyte: byte;
                               var doit: boolean);
var
   timerisrhook:     kbdhooktype;
   kbdisrhook:       kbdhooktype;
```

The procedure named by KBDISRHOOK is called by the A804XDVR ISR whenever a cooked keyboard interrupts with a keystroke. 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 KBDTRANSHOOK has been called, and before any file system-related processing takes place (unless the **Any char** (or 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).

KBDISRHOOK is initialized to the standard keyboard ISR KEYSERVICE in module KEYS. When you assign 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.
- DOIT is passed in as a var 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.

If, upon returning from an ISR, DOIT has been set to TRUE, any subsequent routine in the series of ISR calls will be made. And as mentioned, if DOIT has been set to FALSE, no further processing will take place. For example, in procedure A804XISR (in module A804XDVR), a keypress gets sent to KBDTRANSHOOK, and then, *if* DOIT *is still true after returning,* the keypress also gets sent to KBDISRHOOK.

Analogous remarks apply to timer interrupts.

KBDISRHOOK and RPGISRHOOK Status Byte

Here is the definition of the Status byte when KBDISRHOOK or RPGISRHOOK is called. Please note that the definition is different for other kinds of interrupts.

Bit	Meaning
7	Most significant bit; always equal to 1 for KBDISRHOOK and RPGISRHOOK calls.
6	Always equal to 0 for KBDISRHOOK calls, 1 for RPGISRHOOK calls.
5	Equal to 0 if [Ctrl] pressed, equal to 1 if not.
4	Equal to 0 if [Shift] pressed, equal to 1 if not.
3	Equal to 0 if [Extend char] pressed, equal to 1 if not.
2-0	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(StatByte div 32);
```

Alternatively, you might declare a packed record which has boolean fields in the right places and trick the value of Status into this record:

```
type
  StatusRec=            packed record
                          case integer of
                            1: (ByteValue:      byte);
                            2: (KBDorRPGcall,
                                KnobPulse,
                                NotControl,
                                NotShift:       boolean;
                                Filler:         0..15);
                        end;
var
  TranslateStatus:      StatusRec;
begin
```

```
with TranslateStatus do
  begin
  ByteValue:=StatByte;
  if NotShift then
    begin
      .
      .
      .
```

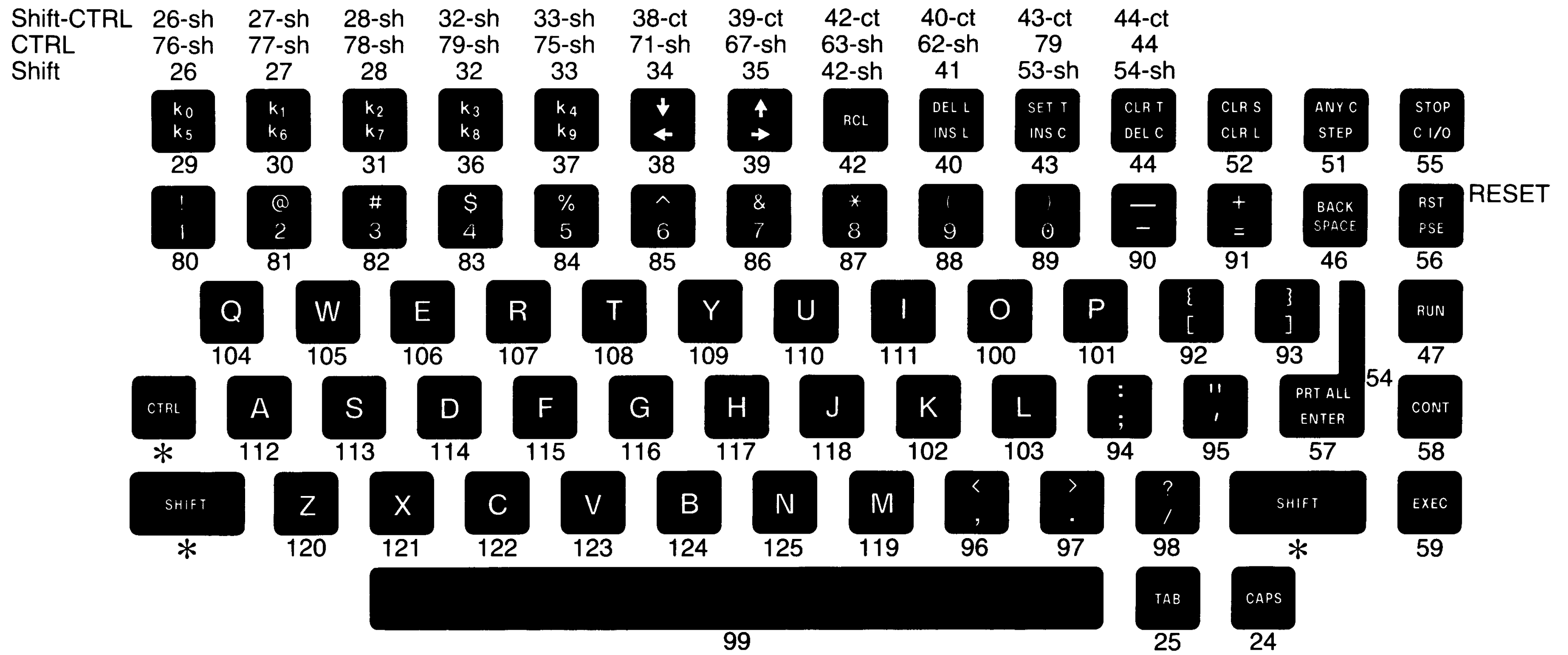
KBDISRHOOK Data Byte

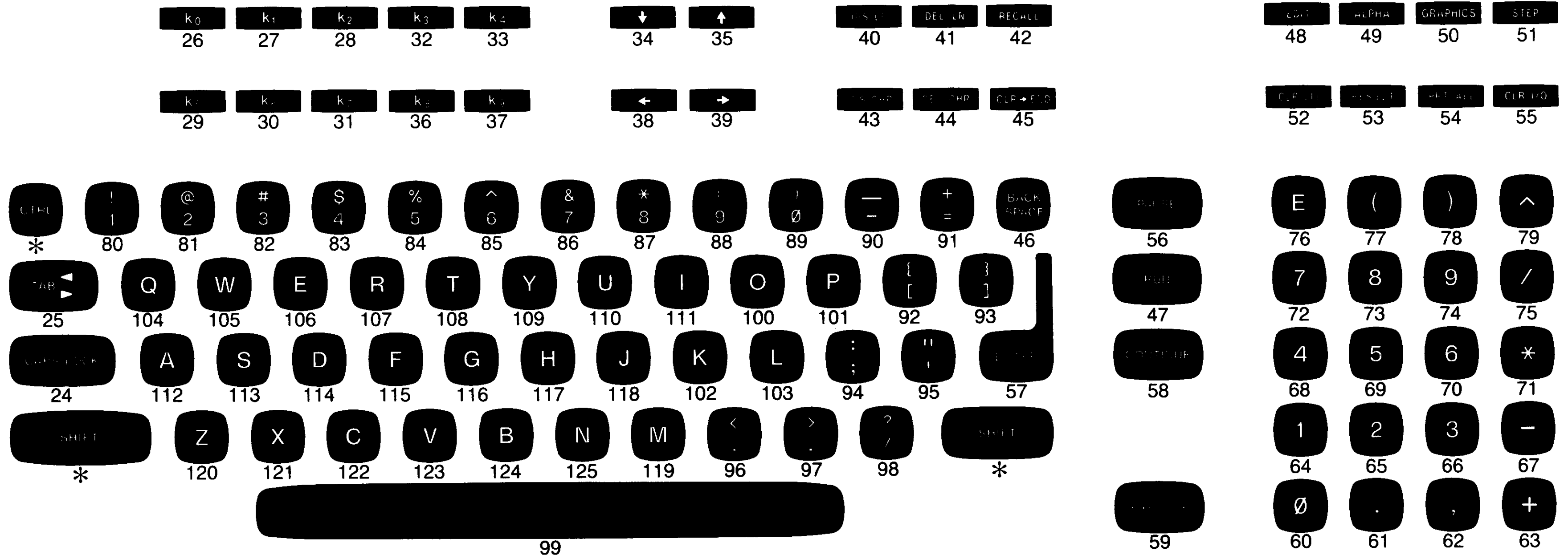
If bit 6 of the status byte equals zero, indicating a keystroke, then the correspondence between DATABYTE values and physical keys pressed on the various keyboards keyboard is given by the pictures on the following three pages. Please note that the "names" of the keys shown are those on the standard English keyboard. If your machine has a foreign keyboard, some of the keycaps may be in different positions.

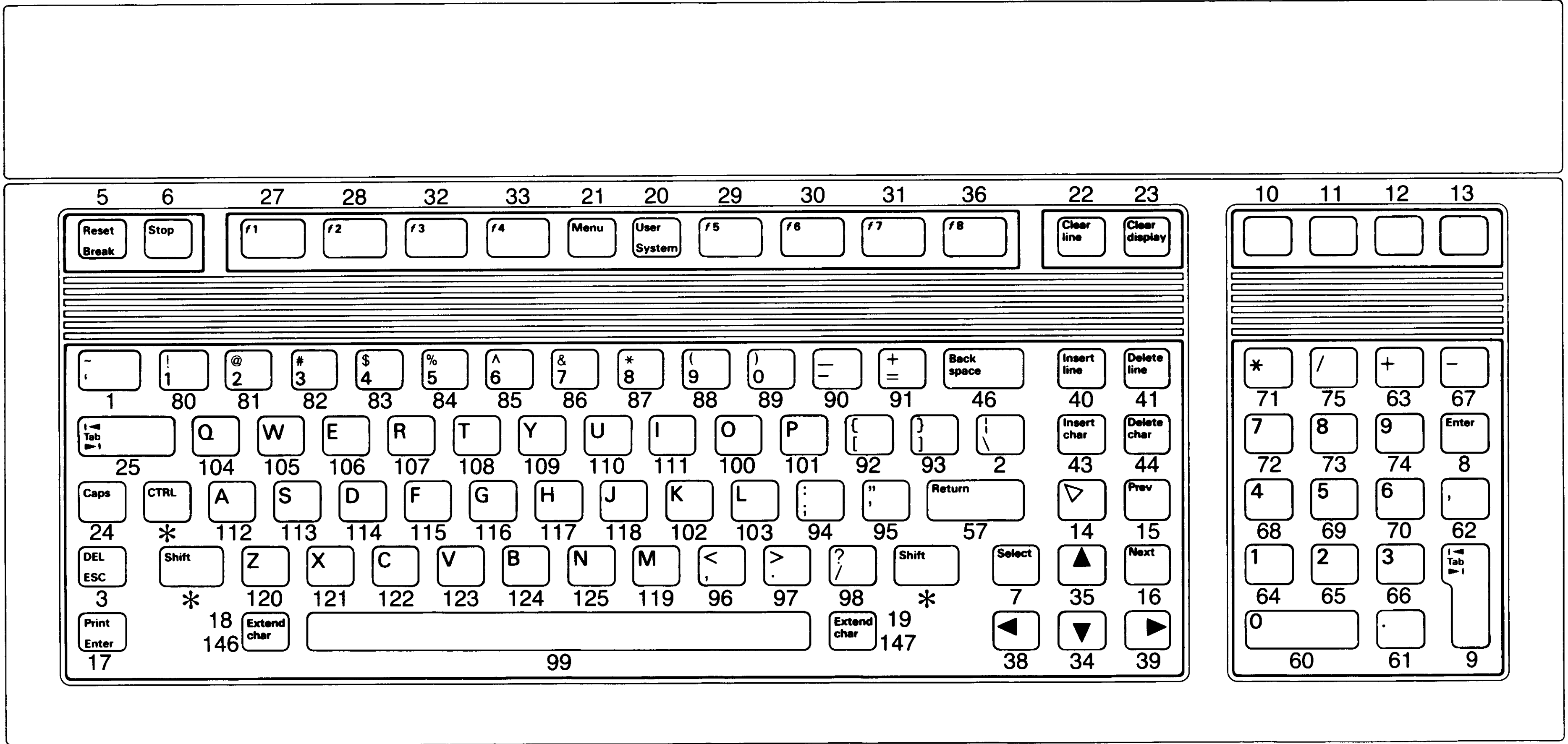
RPGISRHOOK Data Byte

If you are replacing RPGISRHOOK, 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 DataByte<128 then
  Count:=DataByte        {counterclockwise}
else
  Count:=128-DataByte;          {clockwise}
```


Shift-CTRL 26-sh 27-sh 28-sh 32-sh 33-sh 38-ct 39-ct 42-ct 40-ct 43-ct 44-ct
CTRL 76-sh 77-sh 78-sh 79-sh 75-sh 71-sh 67-sh 63-sh 62-sh 79 44
Shift 26 27 28 32 33 34 35 42-sh 41 53-sh 54-sh
k0 k5 29
k1 k6 30
k2 k7 31
k3 k8 36
k4 k9 37
38
39
RCL 42
DEL L INS L 40
SET T INS C 43
CLR T DEL C 44
CLR S CLR L 52
ANY C STEP 51
STOP C I/O 55
RESET
! 1 80
@ 2 81
# 3 82
$ 4 83
% 5 84
^ 6 85
& 7 86
* 8 87
(9 88
) 0 89
— – 90
+ = 91
BACK SPACE 46
RST PSE 56
Q 104
W 105
E 106
R 107
T 108
Y 109
U 110
I 111
O 100
P 101
{ [92
}] 93
54
RUN 47
CTRL *
A 112
S 113
D 114
F 115
G 116
H 117
J 118
K 102
L 103
: ; 94
" ' 95
PRT ALL ENTER 57
CONT 58
SHIFT *
Z 120
X 121
C 122
V 123
B 124
N 125
M 119
< , 96
> . 97
? / 98
SHIFT *
EXEC 59
99
TAB 25
CAPS 24


k0 k1 k2 k3 k4
26 27 28 32 33
34 35
DEL LN RECALL
40 41 42
ALPHA GRAPHICS STEP
48 49 50 51
29 30 31 36 37
38 39
CLR→END
43 44 45
CLR I/O
52 53 54 55
* 80 81 82 83 84 85 86 87 88 89 90 91 46
! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 9 Ø — – + =
BACK SPACE
Q W E R T Y U I O P { [}]
25 104 105 106 107 108 109 110 111 100 101 92 93
A S D F G H J K L : ; " '
24 112 113 114 115 116 117 118 102 103 94 95 57
Z X C V B N M < , > . ? /
* 120 121 122 123 124 125 119 96 97 98 *
99
56 47 58 59
E ( ) ^
76 77 78 79
7 8 9 /
72 73 74 75
4 5 6 *
68 69 70 71
1 2 3 −
64 65 66 67
Ø . , +
60 61 62 63



5 6 27 28 32 33 21 20 29 30 31 36 22 23 10 11 12 13
Reset Break
Stop
f1 f2 f3 f4
Menu
User System
f5 f6 f7 f8
Clear line
Clear display
1 80 81 82 83 84 85 86 87 88 89 90 91 46 40 41 71 75 63 67
Back space
Insert line
Delete line
Tab
Q W E R T Y U I O P
25 104 105 106 107 108 109 110 111 100 101 92 93 2 43 44 72 73 74 8
Insert char
Delete char
Enter
Caps
CTRL
A S D F G H J K L
Return
Prev
24 112 113 114 115 116 117 118 102 103 94 95 57 14 15 68 69 70 62
DEL ESC
Shift
Z X C V B N M
Shift
Select
Next
3 120 121 122 123 124 125 119 96 97 98 7 35 16 64 65 66
Print Enter
Extend char
Extend char
18 146 19 147
17 99 38 34 39 60 61 9

The 98203A Keyboard

The 98203A keyboard must return keycodes that correspond to the equivalent keycap notation of the 98203B keyboard since the physical arrangement of the 98203A keyboard is not the same as the 98203B keyboard. There are some keys on the "B" keyboard that are not available on the "A" keyboard due to the differences in the number of keys. In other cases, some keycaps of the "A" keyboard contain different pairs of functions than the "B" keyboard. Not all functions are shown on the keycap of the "A" keyboard and require the use of `Ctrl` and `Shift`-`Ctrl` to access the equivalent keycodes of the "B" keyboard.

This is a second-revision 98203A keyboard. The keyboard can be identified by reading the keyboard configuration register. Bit 6 is set (1) if the keyboard is the second-revision keyboard, and is reset (0) if the keyboard is the first-revision 98203A keyboard.

The second-revision keyboard returns a different key-code for the `Shift`-`Ctrl` of most of the top row of keys. The `ENTER` key also returns a different key-code when `Shift` and/or `Ctrl` has been pressed.

NOTE

At this time, the 98203A Swedish/Finnish keyboard (option 850) does not have the "*" character. The proposed location for this key is the `Ctrl`-`DEL CHR` position. Depending on the revision of the keyboard, this key may be available.

To see an example of a program which takes control of the keyboard, refer to the *System Devices* chapter of the *Pascal 3.0 Procedure Library* manual.

Gritty Details of the Keyboard

The next several pages present an abbreviated "Internal Reference Specification" of the keyboards with emphasis on matters of interest to a programmer, including a description of all the commands the 68xxx can issue to the 804x. At the end of the section we return to the Pascal 3.0 `A804XDVR` module, which exports three routines which can be used to command the keyboard directly.

About the Electronics

In the Models 226 and 236 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 Model 216, 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 804x 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/real-time 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 ROMs in the keyboard controller.

Circuits Common to the 98203 Keyboards

The RESET Key

The `Reset` (`SHIFT` - `PAUSE` or `SHIFT` - `PSE`) key is scanned like any other key but handled specially by the 804x. Instead of outputting the keycode, the 804x generates an NMI. When the 68xxx'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 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 804x 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.

Note: this discussion of `Reset` is also valid for the 46020A keyboard.

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 98203B keyboard, the 804x receives two derived signals; one says the knob has pulsed, the other tells which direction.

The Beeper (98203A/B Keyboards)

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 804x.

The Beeper (46020A Keyboard)

There is a new, four-voice beeper, or, more accurately, "sound generator," available with the 46020A keyboard. It can produce three independent tones simultaneously, as well as periodic or white noise. See the section *Using the Four-Voice Sound Generator*, later in this chapter.

Protocol for Keyboard Handling

Communication Addresses

Status register:	Read byte at address $428003
Command register:	Write byte to address $428003
Data to CPU:	Read byte at address $428001
Data to 804x:	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 68xxx

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]-[Break] or [SHIFT]-[PAUSE] or [SHIFT]-[PSE]) has been pressed. (There are other sources of NMI than the keyboard.)

When a level seven interrupt is being requested, the 804x 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]-[Break] or [SHIFT]-[PAUSE] or [SHIFT]-[PSE]) was pressed; if one, the interrupt is a timeout. *Note: This is true for hardware supported by Pascal 3.0 but is not guaranteed to be true in the future.*

Bit zero of the keyboard Status register is a one when level one service is desired. The 804x can be commanded to mask any or all its reasons for interrupting. When any capability of the 804x 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 804x is different from masking interrupts in the 68xxx. If the 68xxx is running at level one or higher, and the 804x is enabled to interrupt, the Status register will so indicate. Status can be polled at any time.

Sending a Command to the 804x

To tell the 804x 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 804x's side of the "handshake"; when it is zero, the 804x is ready for the next byte of data or has carried out the command. The three routines `SENDCMD`, `SENDDATA`, and `CMD_READ_1` are exported from module `A804XDVR` to perform the above operations.

Processing an 804x Service Request

This goes on at priority level one. Either the 804x 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.

1. When the sequence starts, read Status to determine the nature of the service request.
2. 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.
3. For each subsequent byte of data that goes with the message, the 804x 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
0000*xxxx*	Not used.
0001*xxxx*	10 msec Periodic System Interrupt (PSI).
0010*xxxx*	Interrupt from one of the special timers.
0011*xxxx*	Both PSI and special timer interrupting.
0100*xxxx*	The Data register contains a byte requested by 68xxx.
0101*xxxx*	HP-HIL status in data register.
0110*xxxx*	HP-HIL data in data register.
0111*xxxx*	Not used.
1000*xxxx*	Data contains key address (both `Shift` and `Ctrl` pressed).
1001*xxxx*	Data contains key address (`Ctrl` only).
1010*xxxx*	Data contains key address (`Shift` only).
1011*xxxx*	Data contains key address (no `Shift` or `Ctrl`).
1100*xxxx*	Data contains knob count (`Shift` and `Ctrl`).
1101*xxxx*	Data contains knob count (`Ctrl` only).
1110*xxxx*	Data contains knob count (`Shift` only).
1111*xxxx*	Data contains knob count (no `Shift` or `Ctrl`).

Data Requests From Within an ISR (Don't do it)

Care should be taken when requesting data from the 804x, particularly from within an ISR, as only one data request can be processed at a time. This can be a problem if a foreground program requests a data byte from the 804x and an interrupt-driven background process initiates a request for data before the foreground process receives the data it requested. Data from one request must be accepted before another request is initiated. If a second data request is made before data from the first is accepted one of the requests may not receive data or (more likely) the data returned may be received out of order.

Care should also be taken when transferring data to the 804x. Data transfers to the 804x all use the same data input pointer. All data to be transferred must be sent before a new command (which changes the data input pointer) can be sent. If, for instance, a foreground process began a multi-byte transfer of data to the 804x which was interrupted by a background process which also transfers data to the 804x, the pointer would be modified by the interrupt process and any further data sent to the 804x by the foreground process would not go to the correct register. Be very careful here, as the results are unpredictable.

Using the Four-Voice Sound Generator

The beeper has been upgraded in HP-HIL keyboards to be a four-voice Texas Instruments SN76494 sound generator: three of the four independent voices produce square wave tone output, and the other is a semi-programmable noise source. Each voice has an independent volume control with 30 dB of dynamic range in 2 dB increments. Each voice also has an independent duration timer (implemented in the keyboard processor) which affords timed durations of up to 2.5 seconds in 10-millisecond increments.

The interface to the sound generator is through the keyboard processor, and all functions take place in the sound generator itself except for the duration timers; these are in the keyboard processor. Data is transferred to the sound generator by writing four bytes to a data buffer and sending a trigger. When the trigger is received, the first three bytes sent to the buffer will be written to the sound generator in sequence. The fourth byte sent to the buffer is used to initialize a duration timer for the appropriate voice.

To write to the data buffer, the data input pointer must be set up by sending the command `$E0`. The beep trigger command is `$C4`. Note that the data buffer is the same as the one used to transfer data out to the HP-HIL interface. The same precautions about sending the data bytes associated with a command in an uninterruptable sequence applies here.

The data format required is rather complex because it includes the addressing information required by the sound generator for register- and voice identification. Three bits of the first byte determines the voice number (2 bits) and the attribute (1 bit) of that voice to be defined: frequency or attenuation. Four more bits in the first byte and six bits in the second byte define the frequency of the tone (10 bits) and the third byte determines the attenuation of the tone (4 bits). All eight bits in the fourth byte specify the tone's duration. The duration timer number (voice association) is determined from the addressing information supplied in the attenuation byte. The frequency of a tone may be calculated by the function $f=83333/n$, where n is the 10-bit number sent in the combination of the two frequency bytes[2].

Note that the voice number is specified twice in the four bytes: once in byte one and again in byte three. This is simply the format required by the TI sound generator. You could, if you wanted to, specify bits 6–4 of byte one as `010` and the same bits in byte three as `101`. This would result in setting the frequency for Voice 2 and the attenuation for Voice 3. In most applications, however, those bits in bytes one and three will be the same.

[2] 333,333 Hz is the frequency of the clock supplied to the part. Internally, this is divided by four, resulting in 83,333 Hz.

Control Registers in the Sound Generator

The sound generator has eight internal registers which are used to control the three tone generators and the noise source. During all the data transfers, the first byte contains a 3-bit field which determines the destination control register. The register address codes are shown below:

R2	R1	R0	Voice
0	0	0	Tone 1 Frequency
0	0	1	Tone 1 Attenuation
0	1	0	Tone 2 Frequency
0	1	1	Tone 2 Attenuation
1	0	0	Tone 3 Frequency
1	0	1	Tone 3 Attenuation
1	1	0	Noise Control
1	1	1	Noise Attenuation

The first two bits determine the voice and the third bit selects between frequency and attenuation.

In the table below, the four bytes required to initiate a tone from one of the voices are detailed. There are four fields defined by the bit designators R2–R0, F9–F0, A3–A0, and D3–D0. In all the following bit designators, the bit labelled "0" (e.g., R0) is the least significant bit.

R2–R0	Register address field, as described above.
F9–F0	Frequency determination field: $f=83333/n$.
A3–A0	Attenuation determination field.
D7–D0	Duration field.

A 1 or 0 in a bit position indicates a mandatory bit value.

Byte 1: (Frequency 1)	*(msb)*	1	R2	R1	R0	F3	F2	F1	F0	*(lsb)*
Byte 2: (Frequency 2)	*(msb)*	0	0	F9	F8	F7	F6	F5	F4	*(lsb)*
Byte 3: (Attenuator)	*(msb)*	1	R2	R1	R0	A3	A2	A1	A0	*(lsb)*
Byte 4: (Duration)	*(msb)*	D7	D6	D5	D4	D3	D2	D1	D0	*(lsb)*

Note that there is no register identification in byte 2; the register used depends on the address field of byte 1.

If the attenuation field is `$0`, the resultant tone will have maximum volume. As the attenuation value is increased, the volume decreases by approximately 2 dB per count. A value of `$F` in the attenuation field turns the tone off.

The duration of a tone is the value of the duration byte multiplied by 10 msec. Hence, tones may be automatically timed from 0.01 seconds to 2.55 seconds in duration. A value of 0 for the duration byte turns the tone on indefinitely; it must be then terminated by the system. The system can determine if any voice has timed out by reading the contents of the voice timer registers. The timers count down from the initial value to zero. A value of zero indicates that the time limit has expired and the voice has been turned off, unless zero was supplied for the duration. The commands to read the voice timer registers associated with each voice number are listed below.

F4	Read the voice #1 timer.
F5	Read the voice #2 timer.
F6	Read the voice #3 timer.
F7	Read the voice #4 timer.

Using the Noise Generator

Only one byte is required by the sound generator to set up the noise generator, however, the interface through the keyboard processor will sent two (bytes one and two). Therefore, when data is written to the noise control register, it should be duplicated in both byte one and byte two of the data input buffer. The noise source also has associated attenuation and duration bytes which are controlled exactly as for the other voices.

The noise source is a shift register with an exclusive-or feedback network. Whenever the noise control register is changed, the shift register is cleared. The shift register will shift at one of four rates as determined by the two NF ("noise frequency") bits, described below. The fixed shift rates are derived from the clock input.

The noise control byte has the following layout:

Noise Control Byte: *(msb)* `1 R2 R1 R0 0 FB NF1 NF0` *(lsb)*

R2, R1, and R0 must, of course, be 1,1,0 respectively to to select the noise control register. The other three bits control the characteristics of the noise generated as follows: if the FB ("feedback") bit is 0, periodic noise is generated, and if FB is 1, white noise is generated. The two bits NF1 and NF0 control the noise generator shift rate as described below:

NF1	NF0	Shift Rate
0	0	$M/64$
0	1	$M/128$
1	0	$M/256$
1	1	Use tone generator 3 output

where M is the clock frequency input: 333333 Hz.

Four-Voice Beeper Example

Below is an example showing how to use the four-voice beeper. The procedures SENDCMD and SENDDATA are exported from module A804XDVR (see *"Black Box" Description of Functions*, next, for further details on these procedures). Voice 2 is used, the specified frequency is $83333/31 \approx 2688$ Hz, the attenuation is minimum, resulting in the loudest tone, and the duration is one second.

```
program Four_voice;
import a804Xdvr;

const
  Voice2F=          '010';                  {R2-R0: voice for frequency}
  Frequency=        '0000011111';           {F9-F0: frequency}
  Voice2A=          '011';                  {R2-R0: voice for attenuation}
  Attenuation=      '0000';                 {A3-A0: attenuation}
  Duration=         '01100100';             {D7-D0: duration}

begin
  sendcmd(hex('E0'));                                  {reset input pointer}
  senddata(binary('1'+Voice2F+str(Frequency,7,4)));    {byte 1}
  senddata(binary('00'+str(Frequency,1,6)));           {byte 2}
  senddata(binary('1'+Voice2A+Attenuation));           {byte 3}
  senddata(binary(Duration));                          {byte 4}
  sendcmd(hex('C4'));                                  {trigger}
end.
```

Command "A3" Beep Compatibility

For reverse compatibility, the beep command $A3 works on the new HP-HIL keyboard controller just it does on as the other keyboard controller ICs used in other Series 200 products. This command causes a tone of the correct frequency and duration to be generated using voice three at maximum amplitude. The frequency of the old beep was given by $f = 81.38 \times n$, where n is a 6-bit number sent by the host system. The new frequencies available are given by the formula $f = 83333/n$, where n is a 10-bit number. This means that the new tone generators is capable of more of a chromatic scale with a greater dynamic range than the old beeper, but is not capable of producing exactly the same frequencies for a given beep command as those which are produced by the old hardware. The frequency generated is the one which is closest to the old beep frequency as possible, given ten bits of control.

"Black Box" Description of Functions

The following descriptions refer to a thing called the "timer output buffer", which is simply five contiguous bytes in the 804x's address space[3]. This buffer can be examined by commands to the keyboard processor.

A command sent to the 804x cannot be considered carried out until bit one of Status is cleared to zero. If 68xxx 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 804x is considerably slower than the 68xxx, and it may be off servicing a timer when the command arrives.

When any command is sent which asks for data, an interrupt will occur in response. This interrupt can't be masked.

Use the A804XDVR procedures as follows:

```
sendcmd(command);
senddata(byte1);
{senddata(byte2);     {use this if two bytes are needed}
{senddata(byte3);     {use this if three bytes are needed}
```

Some commands cause the keyboard processor to return data to the CPU. Use CMD_READ_1 for these as follows:

```
cmd_read_1(command, response);
```

Generalities

When generating a non-maskable interrupt, bit 2 of Status register is set, indicating "not Reset key." This flag bit won't be overwritten even if Reset is pressed before the 68xxx reads the status register, because the Reset key is artificially delayed 1.6 milliseconds from when it goes down. Asserts NMI. NMI will remain asserted until either a system Reset or Cancel command occurs.

There is no command to read the knob counter; the knob must reach the end of its accumulation period and send the data via an interrupt.

The keyboard will interrupt every 10 msec when PSI is not masked. PSI interrupts may occasionally be accompanied by real-time clock interrupts (i.e., bits 1 and/or 0 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 two msec.

All keys auto-repeat except Reset, Shift, Ctrl. Repeating starts after a given key has been down longer than the specified delay.

When maintaining the real time, the 804x updates five bytes. When the three least significant bytes, which count centiseconds since midnight as an integer, reach 23 hours, 59 minutes, 5999 centiseconds, the day (top two bytes) is incremented. Invalid times greater than or equal to 24 hours will be rolled over and become valid within 10 minutes.

[3] The 8041 has a 64 bytes of RAM, and the 8042 has a 128 bytes of RAM.

Load Timer Output Buffer Commands

The timer output buffer commands are commands that the 68xxx can use to read the current values of the cycle interrupt timer, the fast handshake timer, and the real time clock. The output buffer is a bank of five registers that the 804x uses to temporarily store the data that the 68xxx has asked for. Their addresses are `$13` through `$17`.

For these commands, the data may be read by issuing the command sequence `$13`, `$14`, `$15`... for as many registers as needed to get a complete answer.

Sample Time-of-Day

`$31` (49 decimal) Copies the five bytes of real time/date into the timer output buffer. Further commands are necessary to read it.

The first three bytes will contain the number of centiseconds since midnight. The next two bytes contain the number of days since time was set. The least significant byte of the time will be in the first location of the buffer and the most significant will be in the third location. The least significant byte of the day will be in the fourth byte of the buffer and the most significant in the fifth byte. Both the days and and the centiseconds are given as positive-true binary numbers.

Sample Fast Handshake Time

`$36` (54 decimal) Load the timer output buffer with the fast handshake time. The first two bytes of the timer output buffer will contain the current value of the fast handshake timer. The least significant byte will be in the first buffer location and the most significant byte in the second location. See also *Non-Maskable Interrupt* commands `B2` in *Set-Up Commands*.

Sample Match Time

`$38` (56 decimal) Copies the match time into the timer output buffer. The first three bytes of the timer output buffer will contain the match value. The least significant byte will be in the first buffer location and the most significant byte in the third location.

Sample Delay Timer

`$3B` (59 decimal) Copies the three bytes of the current delay timer into the first three bytes of the timer output buffer. The first three bytes of the timer output buffer will contain the current value of the delay timer. The least significant byte will be in the first buffer location and the most significant byte in the third location.

Sample Cycle Timer

`$3E` (62 decimal) Copies the three bytes of the current cyclic timer into the first three bytes of the timer output buffer. The first three bytes of the timer output buffer will contain the current value of the cycle timer. The least significant byte will be in the first buffer location and the most significant byte in the third location.

Data Request Commands

The data request commands are used by the 68xxx to access a byte of data from the 804x. These commands request data from the 804x. The following is a list and description of each of the data request commands.

Request 804x RAM Data

`$00` (0 decimal)	Load the register at `$428001` with the current byte of 804x RAM data and increment the RAM location counter. This will cause an interrupt. (This is a debugging tool.) See also command `$C1`, which resets the 804x RAM pointer.

Request Interrupt Mask

`$04` (4 decimal)	Sends the current interrupt mask back through the register at `$428001`. This will cause an interrupt.

Request Configuration Jumper

`$11` (17 decimal)	Sends back a value indicating which jumpers on the mother board configuration are closed. The other bits in the bytes returned have the meanings specified below (attributes are true if bit is *set* (`1`). This will cause an interrupt.

- HP-HIL Keyboards: Bit 5 of data register is set (`1`).

Bits 0,1,2	Not applicable.
Bit 3	N-key rollover. `1` (true) on present HP-HIL keyboards.
Bit 4	No keyboard on HP-HIL loop.
Bit 6	804x code revision (`0` at present).
Bit 7	ID PROM available in keyboard controller.

- Non-HP-HIL keyboards: Bit 5 of data register is cleared (`0`).

Bit 0	Jumper 3. `0`→98203B keyboard (internal or external); `1`→98203A keyboard.
Bit 1	Jumper 4 (unassigned).
Bit 2	Jumper 5 (unassigned).
Bit 3	Jumper 6 (unassigned).
Bit 4	Jumper 7. Keyboard is not plugged in.
Bit 6	`0`→old ROM code for 804x keyboard processor; `1`→new revision ROM code for 804x (top row control keys, N-key rollover, and ID PROM).
Bit 7	Always `0`.

NOTE

The unassigned jumpers are wired so as to yield a 0 in the associated bit position. The remaining bits 5 through 7 are set to 0.

Read Language Code (HP-HIL only)

$12 (18 decimal) Sends the keyboard language code back through the Data register. This will cause an interrupt. The values whose meanings have been assigned are:

- HP-HIL Keyboard

$03 (0000 0011)	Swiss French
$07 (0000 0111)	Canadian English
$0B (0000 1011)	Italian
$0D (0000 1101)	Dutch
$0E (0000 1110)	Swedish
$0F (0000 1111)	German
$13 (0001 0011)	European Spanish
$15 (0001 0101)	Belgian (Flemish)
$16 (0001 0110)	Finnish
$17 (0001 0111)	United Kingdom
$18 (0001 1000)	Canadian French
$19 (0001 1001)	Swiss German
$1A (0001 1010)	Norwegian
$1B (0001 1011)	French
$1C (0001 1100)	Danish
$1D (0001 1101)	Katakana
$1E (0001 1110)	Latin Spanish
$1F (0001 1111)	United States (USASCII)

- 98203B Keyboard

$00 (0000 0000)	Standard English keyboard.
$01 (0000 0001)	French
$02 (0000 0010)	German
$03 (0000 0011)	Swedish/Finnish
$04 (0000 0100)	Spanish
$05 (0000 0101)	Katakana
$06 (0000 0110)	Jumper J8
$07 (0000 0111)	Jumper J10
$08 (0000 1000)	Jumper J11

- 98203A Keyboard—The values listed above for the 98203B keyboard apply as well to the 98203A keyboard for values 0 through 5. Jumper designations J8 through J11 do not apply. Actually, the implementation on the 98203A keyboard consists of three jumpers: J0, J1, and J2. These are read during the power-up or 68xxx reset first scan of the keyboard and not read again. A cut jumper indicates a "1"; therefore, a Spanish keyboard should have jumper J2 cut and jumpers J0 and J1 uncut, or connected. The remaining jumpers J4 through J7 are used as configuration jumpers.

Read Timer Output Buffer

$13 (19 decimal)	Read timer buffer's least significant byte.
$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 causes a byte to be sent to the Data register and interrupts the 68xxx. These interrupts are not maskable within the keyboard processor.

Request 804x Register

$F*x* (240 through 255 decimal)	Request a byte of data from any one of the top 16 registers in the 804x RAM space. See also the E*x* command, *Write to 804x Register*, in the *Set-Up Commands* section, later in this chapter.

Set-Up Commands

The set-up commands are a group of commands that are used by the 68xxx to initialize or set up the 804x. Use the `SENDDATA` procedure (exported by `A804XDVR`) to handshake the data bytes through the 804x data register. The commands are as follows:

Set Auto-Repeat Delay

`$A0` (160 decimal) Takes one data byte which specifies the delay before the auto-repeat of the depressed key starts. The delay is equal to `bincmp(RATEBYTE)*10+10`. For example: 0→2560 msec delay; 255→10 msec delay.

Set Auto-Repeat Rate

`$A2` (162 decimal) Takes one data byte which specifies the desired repeat rate. If the value is zero, the auto-repeat is turned off; otherwise, the repeat rate is equal to `bincmp(RATEBYTE-1)*10`. For example: 0→no repeat; 1→2550 msec repeat rate; 255→10 msec repeat rate.

Beep (All Keyboard Models)

`$A3` (163 decimal) This takes two bytes of data. The length of time to beep is a byte equal to `bincmp(DURATION)*10+10`; for example, a duration byte value of 0 (binary 00000000)→2560 msec, and 255 (binary 11111111)→10 msec. The frequency byte is less than or equal to 63, where 0 means stop beeping and 63 is the highest frequency. If you are not using the four-voice sound generator, the actual beep frequency in Hz will be $f=81.38\times n$, where n is a 6-bit frequency value. If you are using the four-voice sound generator, the actual beep frequency will be $f=83333/m$, where m is an integer selected such that f is as close as possible to a multiple of 81.38 Hz. In other words, although the four-voice sound generator will generator a tone with a very similar frequency, it will probably not be exactly the same tone as would be generated without the four-voice sound generator.

A beep starts sounding as soon as the second byte of data (the frequency byte) is received. If a beep command is sent while an old beep is still going, the new beep will cancel the old beep. If a beep is going, and you want to stop it, you can send another beep command with a frequency byte whose value is 0.

Set Built-In Knob Pulse Accumulation Period

`$A6` (166 decimal) The built-in knob accumulates pulses for a specified period before interrupting the 68xxx to report the count. This command specifies the period with one data byte. The duration of the period in milliseconds is defined as `BYTE*10`, unless the value of `BYTE` is zero, in which case the duration is 2560 msec. For example, a byte value of 1→10 msec period; 255→2550 msec period; 0→2560 msec period.

The built-in knob is normally scanned every 100 μsec, 200 μsec when a key is detected, and 500 μsec when a 10 msec interrupt occurs.

This is essentially a no-op as far as the HP-HIL keyboard is concerned because all locators return raw HP-HIL information with a 20 millisecond rate.

Set Time-of-Day and Date

$AD (173 decimal) Set up to input the real time in centiseconds. Begins counting time from the moment the 804x begins executing the command. Follow with three or five bytes of data, least significant byte first. The first three bytes are the number of centiseconds since midnight. The day count may be sent following the time data without sending the $AF command. Last two bytes, if sent, are an integer representing the "day" in whatever calendar you prefer.

Set Date

$AF (175 decimal) Follow with two bytes, least significant byte first, just like the optional last two bytes of the $AD command.

Set up Non-Maskable Timeout

$B2 (178 decimal) Set up to input a fast handshake delay. Takes two bytes, LSB first, indicating the number of centiseconds until an interrupt is generated. This interrupt will be on the NMI line. The value is twos-complemented, so 0→633560 msec and 65535→10 msec. This command cancels any pending non-maskable timeout immediately. Begins counting time after receipt of second byte. If this command is not followed by two bytes of data the fast handshake interrupt will be cancelled (see *Cancel Non-Maskable Timeout*, below).

The designation "non-maskable" refers to the fact that the interrupt generated is on level seven, and the 68xxx cannot ignore it. This timeout is, in fact, maskable in that the 804x can be told to keep it quiet for a while.

Cancel Non-Maskable Timeout

$B2 (178 decimal) Same as setting up a non-maskable timeout, except no data bytes follow. Also releases NMI if it was asserted. 68xxx has to wait a certain interval of time to be sure NMI was released. That interval is defined thus: First, wait until the IBF flag ("input buffer full"—bit 1 of the status register) is zero. Then, wait another 200 μsec.

Set Real-Time Match

$B4 (180 decimal) Set up to input a real time match interrupt. This takes three bytes of data, so match is within 24 hours. Least significant byte first, the three bytes comprise the number of centiseconds since midnight at which to match. Cancels itself after it occurs.

Sending an invalid time (e.g., 36 o'clock) will result in no match unless the clock is also invalid. The 804x will begin checking for the match after it receives the third byte. Note: Any real time match interrupt should be cancelled when the real time is about to be changed, because if you set the time while a real-time match is active, spurious match interrupts may be generated. That is, if you are halfway through changing the real time, and a match takes place, an interrupt will occur.

Cancel Real-Time Match

$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.

Set Delayed Interrupt

$B7 (183 decimal) Set up to input a delay interrupt. Takes three bytes representing delay period in centiseconds, LSB first. Immediately cancels any pending delayed interrupt. Begins counting time after receipt of third byte. Note: the time sent is complemented; all 0s yields the longest delay, all 1's the shortest. The result of shortest could be anywhere between 0 and 10 msec.

Cancel Delayed Interrupt

$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.

Set Cyclic Interrupt

$BA (186 decimal) Set up to input a cyclic interrupt. This command is followed by three bytes, least significant byte first, that is the 2's complement of the cycle time. Time is counted after the third byte is received. If this command is not followed by three bytes of data, the interrupt will be cancelled.

Cancel Cyclic Interrupt

$BA (186 decimal) This is just a Setup command followed by no data. Can be sent at any time. Erases any pending but masked interrupt, etc.

Reset 804x RAM Pointer

$C1 (193 decimal) Reset the RAM data output pointer associated with the $00 command (this is a debugging command). See the section *Data Request Commands*, earlier in this chapter.

Write to 804x Register

$E*x* (224 through 239 decimal) Set up to write one of the top 16 registers in the 804x RAM space. See also the F*x* command, *Read to 804x Register*, in the *Data Request Commands* section, earlier in this chapter.

Trigger Commands

The trigger commands initiate some action, and an interrupt may or may not be generated as a result; they are command specific. The "buffer," referred to below is the 16-byte 804x RAM area in `$E0` through `$EF`. The trigger commands are listed below:

Four-Voice Beeper (8042 only)

`$C4` (196 decimal) — Trigger command to transfer four bytes of data from the buffer to the beeper IC and initiate a "new" beep from one of the voices. For further information, see the section *Using the Four-Voice Sound Generator*, earlier in this chapter.

Trigger Data for HP-HIL

`$C5` (197 decimal) — Trigger command to transfer three bytes of data from the buffer to the HP-HIL interface.

Mask Interrupts

`01`*xxxxxx* binary — Bit 5 (the first *x*) of the mask definition is presently undefined, but should be sent as a 0. Bits 4 through 0 of the mask definition each correspond to some excuse for the 804x to interrupt. Setting a bit to 1 will suppress that interrupt when its condition arises.

Bit 4	Mask the "non-maskable timer interrupt".
Bit 3	Mask the 10 millisecond periodic system interrupt.
Bit 2	Mask the timer interrupts.
Bit 1	Mask the `Reset` key.
Bit 0	Mask keyboard, knob and HP-HIL.

To control interrupt masking, use the `MASKOPSHOOK` with the mask constants from `SYSDEVS`. For example:

```
call(maskopshook, kbdmask+timermask {enable}, resetmask {disable});
```

Where `MASK` constants are powers of two. Because they are powers of two, addition is possible for multiple specifications. See the *Pascal 3.0 Procedure Library* manual, the section on *Enabling Interrupts* for further details.

Keyboard Command Processing

Naturally there's a pattern buried in the preceding description. All the commands that involve getting information from the keyboard/real-time clock are specific cases of just two commands. The 8041 has 64 bytes of on-board RAM, the 8042 has 128 bytes, and every byte can be read by the 68xxx.

Send the command `000`*xxxxx* to address any byte in the lower 32 bytes of 804x memory (*xxxxx* is a number between zero and 32). The command will answer by an interrupt and the Data register.

Send the command `001`*xxxxx* to cause five bytes from the upper 32 to be copied down into the timer output buffer, which can then be read by `000`*xxxxx*. Note that 1*xxxxx* is the highest location to be loaded; the lowest location is 1*xxxxx*–4.

Now here is the use of every byte of 804x memory:

`$0–$1`	Scratch and pointer registers.
`$2`	These bits are a set of flags:
0	Equal to 1 when the loop is in reset mode configuration process.
1	Internal "loop busy" bit. This is used by the 804x internal software, and should not be used by 68xxx programs. This is not always exactly in sync with Bit 2, next.
2	External "loop busy" bit. This is the busy bit to check from 68xxx software.
3	Equal to 1 when the non-maskable timer is in use.
4	Equal to 1 when the cyclic timer is in use.
5	Equal to 1 when the delay timer is in use.
6	Equal to 1 when the match timer is in use.
7	Equal to 1 when the beeper is on.
`$3`	Scratch.
`$4`	Another set of flag bits:
0	Equal to 1 when keyboard, knob, and HP-HIL interface interrupts are masked.
1	Equal to 1 when `Reset` is masked.
2	Equal to 1 when user timer interrupts are masked.
3	Equal to 1 when 10 millisecond PSI is masked.
4	Equal to 1 when fast-handshake interrupt is masked.
5	Equal to 1 when it is time to auto-repeat the current output key.
6	Equal to 1 when the first time repeat delay has elapsed.
7	Equal to 1 when the current output key has repeated at least once.

`$5`	More flag bits:
0	Equal to 0 when either `Shift` key is down.
1	Equal to 0 when `Ctrl` key is down.
2	Equal to 1 when it is time to do a PSI.
3	Equal to 1 when time to do user timer interrupt.
4	State of the *left* shift key; zero means pressed (46020A only).
5	Equal to 1 when it is time to send 68xxx something it asked for.
6	State of the *right* shift key; zero means pressed (46020A only).
7	Equal to 1 when time to do fast-handshake interrupt.
`$6`	FIFO control and miscellaneous:
0	Equal to 1 when HP-HIL poll delay has elapsed.
1	Equal to 1 when the HP-HIL control chip has data.
2	Equal to 1 when HP-HIL is in configuration mode.
3	Equal to 1 when HP-HIL is in fault detected mode.
4	Equal to 1 when a poll was sent as the last HP-HIL command.
5	Equal to 1 when indicates FIFO is *not* empty.
6	Equal to 1 when indicates FIFO is full.
7	Equal to 1 when the loop timed out.
`$7`	Keycode of the current output key; zero if no key is down.
`$8`–`$F`	The 804x's stack space.
`$10`	Reset debounce counter.
`$11`	Configuration jumper code.
`$12`	Language jumper code.
`$13`–`$17`	The five bytes of timer output buffer. Least significant byte is `$13`.
`$18`–`$19`	Scratch and pointer.
`$1A`	Scratch, accumulator storage during interrupts etc.
`$1B`	Timer status bits:
0–4	The number of cycle interrupts missed.
5	Equal to 1 when a cycle is up.
6	Equal to 1 when a delay is up.
7	Equal to 1 when there is a real-time match.
`$1C`	Current knob pulse count (non-HP-HIL).
`$1D`	Pointer register to put data sent by 68xxx.

$1E	Pointer register for data to send to 68xxx.
$1F	"Six-counter" for 804x timer interrupt. Since the input frequency of 333333 Hz does not break down into exact 10-millisecond intervals, pulses are counted, and after every sixth one, a "leap period" is added to maintain accuracy in the elapsed time.
$20	Auto-repeat delay.
$21	Auto-repeat timer.
$22	Auto-repeat rate.
$23	Beep frequency (8041).
$24	Beep timer (counts up to zero).
$25	Knob timer (non-HP-HIL).
$26	Knob interrupt rate (non-HP-HIL).
$27	If bit 6 is a one, the 804x timer has interrupted.
$28–$2C	Not used.
$2D–$2F	Time-of-day (three bytes); least significant byte is $2D.
$30–$31	Days integer (two bytes); least significant byte is $30.
$32–$33	Fast-handshake timer (two bytes); least significant byte is $32.
$34–$36	Real time to match (three bytes); least significant byte is $34.
$37–$39	Delay timer (three bytes); least significant byte is $37.
$3A–$3C	Cycle timer (three bytes); least significant byte is $3A.
$3D–$3F	Cycle timer save (three bytes); least significant byte is $3D.

The following 64 bytes are present only in 8042 keyboard processors. The values in these registers (as well as the first 64 registers) can be retrieved by sending the command $C4, which resets the data output pointer, followed by $00 which loads the data buffer with the current byte of 8042 data, and increments the pointer.

$40–$67	These bytes are used as a FIFO buffer.
$68	FIFO read pointer.
$69	FIFO write pointer.
$6A	HP-HIL timeout counter.
$6B	State counter for configuration state machine.
$6C	Temp storage.
$6D	HP-HIL watchdog timer.
$6E	Temp storage.
$6F	$AA if self-test completed.

$70–$73	Four bytes used as a data buffer for four-voice beep and HP-HIL data.
$74–$77	Four bytes used for voice (beep) timers; $74→Voice #1.
$78	Not used.
$79	KBDSADR keyboard address map.
$7A	LPSTAT HP-HIL interface loop status byte.
$7B	LPCTRL HP-HIL interface loop control byte.
$7C	HP-HIL poll fault counter.
$7D	HP-HIL reconfiguration counter.
$7E	Extended configuration/ID register.
$7F	$55 if self-test completed.

In the case of an NMI (a level 7 interrupt), the 804x status register must be read to determine the type of the NMI. The following is the definition for the status register:

*xxxxx*0*xx*	The NMI was generated to indicate that the Reset key was pressed.
*xxxxx*1*xx*	The NMI was a fast-handshake interrupt.

It is possible to tell if the 804x generated an NMI by reading location $478005 and checking bit 1. If this bit is a one, the 804x has pulled on NMI.

For timer interrupts, the data buffer is defined as follows:

Bits 0–4	If non-zero, an error occurred. This contains the number of cycle interrupts that were missed. Any combination of bits 5, 6, and 7 can be set.
Bit 5	If set, a cyclic interrupt occurred.
Bit 6	If set, a delay interrupt occurred.
Bit 7	If set, a time match interrupt occurred.

The 804x may be used as a polled device rather than an interrupting device by checking bit 0 of the status register. When this bit is one, it means that the 804x is interrupting.

Before a command or data can be written to the 804x, the status register must be read to determine if the input buffer full (IBF) flag, bit 1, is zero. If bit 1 is one, the transfer cannot take place.

When data is asked for, an interrupt will occur. This interrupt cannot be masked.

When a command is sent to the 804x, it cannot be considered carried out until the IBF flag (bit 1 of the status register) has been cleared.

Resetting the 804x

There are two types of reset recognized by the 804x: the power-up reset and the "hard" reset. The following occurs at power-up:

1. A self-test is performed to verify the operation of the 804x.
2. The output FIFO is cleared (8042 only).
3. All voices of the sound generator are turned off.
4. All timers are turned off.
5. The real-time clock is cleared.
6. The key auto-repeat rate is set to infinite.
7. The interrupt mask is set to `$1F` (0001 1111).
8. The HP-HIL interface is configured (8042 only).
9. The `LPCTRL` register is set to `$17` (0001 0111) (8042 only).
10. The first keyboard is identified (8042 only).
11. The language and configuration registers are set up.
12. The poll fault and reconfiguration counters are cleared (8042 only).
13. The self-test result byte is output to the host.

If the HP-HIL interface does not configure, or no keyboard is found, the configuration code register will contain `$30`, indicating no keyboard present, and the language register will contain `$1F`, indicating US ASCII. `LPSTAT` may be read to determine the integrity of the interface. For an 8042, the time between the power-up reset and the self-test result being returned to the system may be as long as 300 milliseconds. This time is determined by the maximum self-test time for a device on the HP-HIL interface (200 milliseconds). For an 8041, the maximum time is around 100 milliseconds.

A "hard" reset of the 804x will result in the following:

1. The output FIFO is cleared (8042 only).
2. All voices of the sound generator are turned off.
3. All timers are disabled.
4. The interrupt mask is set to `$1F` (0001 1111).
5. The `LPCTRL` register is set to `$17` (0001 0111) (8042 only).
6. The self-test result byte is output to the host.

The 8042 determines whether or not the reset is the result of a power-up based on the contents of a few registers. If any one of these registers contains an incorrect code, the entire power-up sequence is executed. This may be forced by writing zero to the "self-test result byte" (command: `$EF`; data: `$00`) and then pulling down the reset line. Note that this will clear the real-time clock. Do not try to do this if bit 5 of the configuration register is 0, as this implies an 8041.

In order to make sure that the 804x does not lose the real time when the 68xxx does a reset command, the following must be done.

1. Send command $31 to the 804x. As usual, do not send the command until the IBF flag is clear.
2. Wait for the command to be taken by checking the IBF flag (bit 1 of the status register must be zero).
3. Do the reset command within 100 μsec.

When the 804x is reset it will pull on interrupt level 1 and will continue to pull on it until about 20 μsec after reset is released. This interrupt should not be serviced. Level 1 interrupts should not be enabled during a reset command and for 20 micro-seconds after the reset command. The status register will not indicate that the 804x is interrupting at this time.

The language or configuration code registers may be read by the host at any time, but they are read from the keyboard only at power-up or after loop reconfiguration.

Knob and Timer Details (8041)

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 twos-complement signed byte, which is negative for clockwise turns and positive for counterclockwise. This is probably counter-intuitive.

The Keyboard at Power-up and Reset

The 804x powers up in the following state:

- The auto-repeat is random.
- The knob interrupt rate is random.
- The real time is random.
- All the 804x interrupts are masked.

The 804x 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 804x interrupts on level one, sending $7 in the upper half of the Status register and $8E in the Data register.

When the 804x'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 μsec after the reset line goes away.

Pascal Interface to the Keyboard

Modules SYSDEVS and A804XDVR export 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.

Following are examples of how to use the three procedures SENDCMD, SENDDATA, and CMD_READ_1, which are exported from module A804XDVR. For instance, say you want to send the keyboard the command to beep. You could say:

```
sendcmd(hex('A3'));
senddata(DurationByte);
senddata(FrequencyByte);
```

As an example of reading data back from the keyboard, suppose you wanted to determine what the knob rate was:

```
sendcmd(hex('26'));
cmd_read_1(hex('17'), Data);
```

804x Code Revision and Features Identification

Four code revisions of 804x peripheral processors now exist for series 200 products. A read of the configuration register (command: $11) will cause data sufficient to identify the part to be returned. In this data byte bits 5 and 6 indicate the part revision. The four part types, their usage and identification is as follows: (part number, bit 6, bit 5, usage).

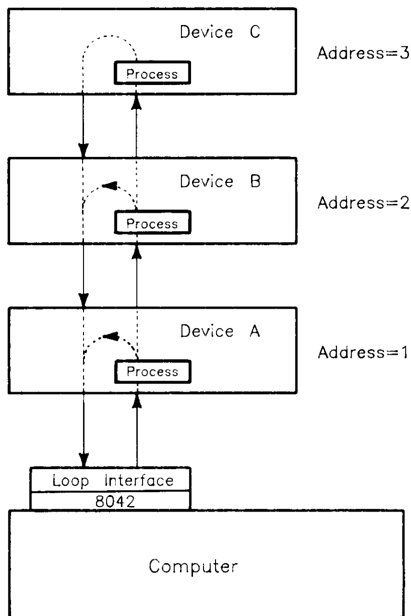
1820-2564 (00)	Used in the Model 26 and Model 36 mainframes.
1820-3087 (00)	Used in the Model 16.
1820-3300 (10)	Used in the Model 16 and on the '34 board for the Model 20.
1820-3712 (01)	Used in the Model 17 and on the '35 board for the Model 20.

The first two parts are identical in operation and no external distinctions need be made. If bit 5 is set there is an extended ID register which can be read to extract more information about implemented features. For the part described in this document it is sufficient to note that bit 5 being set indicates an HP-HIL type of interface.

The Interface to HP-HIL Devices

This section describes communication with the HP-HIL interface, and other functions provided by the 8042 peripheral processor in series 200 products. In fact, the interface to the timers, real time clock and beeper have not changed at all, other than the enhancements. The interface to the keyboard during power up is also nearly identical to the old definition with the only exceptions being changes in the keycodes returned. This is a result of the new 46020A keyboard layout. The following is primarily a description of the enhancements added to handle the HP-HIL interface. This interface is capable of supporting up to seven devices, such as graphics input, system ID PROMs, beepers, and other peripherals generally related to human input, as well as the system keyboard.

Before launching into an in-depth discussion of the workings of the HP-HIL interface, we will present a general overview. HP-HIL stands for "Hewlett-Packard Human Interface Loop," and is, obviously, a *loop*. The following diagram illustrates the basic components.



HP-HIL initialization takes place in the following manner. When the peripheral devices are powered up, they all assume "loopback mode." This means that every device will send all signals received back through the return side of the loop. Thus, every device acts like the "last" device on the loop.

The computer then sends out a signal, looking for the first device on the loop. In our diagram, it would find Device A. Being the first device on the loop, its address is considered to be 1. The computer then instructs Device A to *exit* loopback mode; that is, send the signals through to a possible next device. The computer then attempts to contact the second device on the loop. Device B responds and is assigned address 2. The computer now knows that there are at least two devices on the loop. The computer commands Device B to exit loopback mode, and attempts to contact the next device. Successful, the computer now knows about Device C. As our diagram illustrates, Device C is the last device on the loop, so the process proceeds differently at this point.

The computer instructs Device C to exit loopback mode, and attempts to contact (nonexistent) Device D. Since it is not there, a timeout occurs, and the computer deduces that Device C is the last device on the loop. Therefore, it instructs Device C to once again enter loopback mode, and the loop is configured.

The loop can deal with a maximum of seven devices at any one time. If there are eight or more physically connected, the devices after number seven are not found.

HP-HIL device configuration proceeds until a device is either nonexistent or not powered up. For example, in our diagram above, if Devices A and C were powered up, but Device B was not, the computer would find neither Device B *nor* Device C.

As the above discussion indicates, the address of a particular device is merely its topological order of placement along the loop. In the above diagram, Device A has address 1, B has address 2, and C is address 3. *This is only a result of their physical order of connection.* If Device C had been connected between Devices A and B, Device A would still have been address 1, but Device C would be address 2, and B would be address 3. The type of device is irrelevant to the address assigned to it.

After the loop is operational, and during subsequent loop operations, each device looks at the data being sent down the loop. If a device notices that the destination address associated with the loop data is the same as that device's address, that device receives and acts on the data. Otherwise, the data is merely shuttled along to the next device.

As mentioned earlier in the chapter, if bit 5 of the configuration register is set, there is an extended ID register which can be read to extract more information about implemented features. For the 8042, it is sufficient to note that bit 5 being set indicates an HP-HIL type of interface.

The Interface

Two new status codes have been implemented to handle data transfers from HP-HIL devices to the system. Also new commands have been added to handle data transfers from the system to HP-HIL devices. These will be described below. The 8042 takes care of configuring the HP-HIL interface and error detection as well as some error correction. The 8042 will generally have control of the loop and will be doing the required polling for data. The transfers from the 8042 to the system will, in general, be on an interrupt basis. When data is returned from the loop in response to a poll all of the data is read from the HP-HIL control chip and put in an 8042 internal FIFO stack before the host system (68xxx) is interrupted. Data transfers from this FIFO to the 68xxx take approximately 150 μsec per byte on the Pascal Workstation 3.0 (50 for the host to respond to the interrupt and 100 for the 8042 to get the next data byte out).

When the 8042 generates an interrupt with a status code of `$5`x, the data register contains HP-HIL status information partially describing what caused the interrupt, including, if appropriate, the loop address of the device whose data follows. If bit 3 is set, the first interrupt to follow with status code of `$6`x will have associated data being the command which caused the `$5`x status interrupt.

Status Code 5X

When the 8042 generates an interrupt with a status code of $5x (called a "Status 5 interrupt"), the bits in the Status 5 data byte are defined as follows:

If Bit 7 is clear then:

Bits 0–2	Contain the address to be associated with the data to follow.
Bit 3	When set indicates that the data to follow is the command which caused the previous data bytes (if any) to be returned. This bit may be used as a data packet termination indicator.
Bit 4	Indicates that the data to follow was returned in response to a poll command initiated by the 8042 in auto-poll mode.
Bits 5, 6	Undefined.

If Bit 7 is set then there is some type of exception or error. The remainder of the byte should be interpreted as an "error code" defined as follows:

$80	Indicates that the loop has reconfigured successfully—reconfiguration is complete.
$81	Indicates that there was a parity error, framing error or FIFO overflow in response to a system initiated data or command transfer to the HP-HIL interface. No data will follow.
$82	Indicates that there was a loop timeout in response to a system-initiated data or command transfer to the HP-HIL interface with a timeout interval setup. No data will follow.
$84	Indicates that the loop is being reconfigured. No data will follow.

Status Code 6X

When the 8042 generates an interrupt with a status code of $6x (called a "Status 6 interrupt"), the data register will contain data received (in sequence) directly from the HP-HIL interface. This data is to be associated with the most recently received information byte which was returned with a Status 5 interrupt.

The HP-HIL information byte will be sent at the beginning and end of each data packet as well as whenever the data within a packet is to be associated with a new device address. The data packets are of variable length as defined by the HP-HIL interface protocol.

Whenever poll data (from a raw mode, or "uncooked" device) is received from the HP-HIL interface, the 8042 will interrupt the 68xxx with a Status 5 interrupt and send the data packet. No commands are required from the 68xxx for this transfer. A data stream for a typical packet is shown below. This packet is the result of a mouse at loop address 2 sending X-Y data and a raw-mode keyboard at loop address 4 sending key data, both in the same poll interval.

Status	Data	Meaning
`$5`*x*	`$12`	Poll data from device 2
`$6`*x*	`$02`	Header, two coordinates returned
`$6`*x*	`$00`	X delta (0)
`$6`*x*	`$FF`	Y delta (−1)
`$5`*x*	`$14`	Poll data from device 4
`$6`*x*	`$40`	Header, keycodes, no coordinate data
`$6`*x*	`$5A`	The "A" key went down
`$5`*x*	`$18`	Packet terminator
`$6`*x*	`$15`	Poll command (5 bytes returned)

The HP-HIL interface may be reconfigured (automatically) for several reasons, most of which are listed below:

- An uncorrectable loop error is detected, such as the loss of a poll command or a double parity fault. The most probable cause for this might be electrostatic discharges to loop devices.
- A device is removed from the loop by the user.
- A device is added to the loop by the user.
- The 68xxx requests a loop reconfiguration.

The result of a loop reconfiguration is as follows:

- The language code of the first keyboard found is put in the language code register.
- The `KBDSADR` byte is updated to indicate the current location of all keyboards on the loop.
- Any auto-repeat in process is cancelled.
- The shift and control keys are assumed up.
- The `LPSTAT` register is updated.
- The reconfigure counter is incremented.
- If no keyboards are found the language code and configuration code registers are not changed from their power up values.

New commands have been added to allow the 68xxx to directly read or write the *top* 16 bytes of R/W memory in the 8042. This space contains registers for control and status of the HP-HIL interface as well as buffer space for data transfers. The layout of the space is shown below:

Address	Function
`$70–$73`	General purpose data buffer.
`$74–$77`	Timers for the four voices of the sound generator
`$78`	Unused
`$79`	`KBDSADR`—Bit per address of "cooked" keyboards
`$7A`	`LPSTAT`—HP-HIL interface status
`$7B`	`LPCTRL`—HP-HIL interface control
`$7C`	Poll fault counter (for testing)
`$7D`	Reconfigure counter (for testing)
`$7E`	`REVID`—Code revision/identification byte
`$7F`	Self-test result byte

The command to read one of the top 16 bytes in the 8042 is `$F`*x* where *x* is the byte address (0 to 15). An interrupt with status code `$4`*x* will be generated indicating the data byte is available. The command to write one of these locations is `$E`*x* followed by a write to the data register with the data byte. The address for data input to the 8042 is automatically incremented after a data byte is received, so a buffer may be loaded by sequentially sending data after the initial address setup command. Of course, the normal handshaking procedure must be observed throughout all the transfers from the 68xxx to the 8042.

LPCTRL

The register `LPCTRL` is used to control the operation of the HP-HIL interface and is layed out as follows:

Bit 0 — **Auto-poll:** When set, the 8042 will automatically poll the loop. Data will be returned to the 68xxx only if a device on the loop returns data in response to a poll. The 68xxx may disable auto-polling by setting this bit to a zero. The auto-poll rate is fixed at 50 Hz (20 millisecond intervals).

Bit 1 — **Don't Report Loop Errors:** When set, the 8042 will not inform the 68xxx of parity, framing, FIFO overflow or loop timeout errors discovered after a 68xxx-initiated data transfer. This bit should normally be set the same as the auto-poll bit so the 8042 will handle errors while in auto-poll mode. If the 8042 is not in auto-poll mode and this bit is a one, any error discovered will result in a reconfiguration of the loop. In general, the 68xxx should handle loop errors if the 8042 is not in auto-poll mode.

Bit 2 **Don't Report Loop Reconfiguration:** When set, the 8042 will not inform the 68xxx if the loop is, for some reason, reconfigured. The loop will be reconfigured if an unrecoverable error is discovered while the 8042 is handling loop errors. If this bit is a zero, the "loop being reconfigured code" (status $5*x*; data 84) and "loop reconfigured code" (status $5*x*; data 80) will be sent to the 68xxx at the beginning and end of loop configuration respectively.

Bit 3 **Unused**

Bit 4 **Cook Keyboards:** When set, data from selected keyboards on the loop will be intercepted and translated to emulate the "old" series 200 keyboard keycodes and data transfer mechanism (cooked mode), and data from keyboards which are to be cooked will be returned from those keyboards through `KBDISRHOOK`. If this bit is set to 0, data from all devices on the loop will be returned, without modification, through `STATUS5HOOK` and `STATUS6HOOK` (raw mode).

Bit 5 **Unused**

Bit 6 **Unused**

Bit 7 **Reconfigure the Loop:** The system can cause the loop to be reconfigured by setting this bit. The 8042 will reset the bit and proceed to reconfigure the HP-HIL interface loop. Note: In this case the "loop reconfigured" codes will not be sent.

LPSTAT

The register `LPSTAT` is used to determine the status of the HP-HIL interface and is layed out as follows:

Bits 0–2 Contain a count of the number of devices on the loop.

Bit 3 When set, bit 3 indicates that the loop has been successfully configured. This bit is set to 0 at the start of loop configuration.

Bits 4-6 Unused.

Bit 7 When set, bit 7 indicates that reconfiguration was attempted but not accomplished. Reconfiguration attempts will continue indefinitely until the loop configures (at least one device is found).

KBDSADR

Each bit in the KBDSADR register is associated with a keyboard at a specific address on the loop. If bit 1 is set, there is a keyboard at address 1, if bit 7 is set, there is a keyboard at address 7, etc. This register is used by the 8042 to determine which keyboards get data mapped ("cooked" keyboards) and which have data passed straight through as "raw mode" HP-HIL devices. At power-up or any time the loop is reconfigured, all keyboards on the loop are identified. The KBDSADR register is set up to reflect this configuration by setting corresponding KBDSADR bits to "1". Keyboards are HP-HIL devices whose IDs are in the range 160 to 255. The 68xxx can clear bits in this register to allow "raw mode" access to individual keyboards. Note, however, that a reconfiguration of the loop for any reason will rewrite the register with current configuration information. Because of this, if a system is using some keyboards in mapped mode and some in raw mode, the reconfiguration information transfers should be enabled and the 68xxx must take appropriate action upon loop reconfiguration. The bit in LPCTRL which converts *all* keyboards on the loop to "raw mode" is not changed if the loop is reconfigured.

REVID

The bit pattern in the REVID register can be considered to be an extension of the configuration register (at address $11). It can be read by the command $EA as described previously. The bits are defined as follows:

Bits 0–2	Code revision ID.
Bit 3	Four-voice TI SN76494 beeper available.
Bit 4	Undefined (defaults to 1).
Bits 5-7	Undefined (default to 0).

Note: do not try to read this register if bit 5 of the configuration register is 0.

68xxx-to-HP-HIL Data Transfers

The HP-HIL protocol does not allow more than one command on the loop at a time. As a result, data transfers from the 68xxx to the loop must adhere to a strict handshake protocol. There is a "Loop busy bit" (bit 2 of $02 which indicates to the 68xxx that the 8042 has a command on the loop. Data must not be transferred from the 68xxx to the loop when this bit is set. Also, because the 8042 will put a poll command out every 20 milliseconds when in auto-poll mode, either auto-polling must be disabled or one must be certain of the timing of the data transfer out to the loop so as not to interfere with the polling. When auto-polling is disabled and error reporting is turned on, the 68xxx has complete control of the HP-HIL interface.

Data is transferred to the loop by writing three bytes to a data buffer and sending a trigger. The third byte will be used to initialize a timeout interval (timed by the 8042) so that the 68xxx may recognize a loop timeout. If this last byte is a 0, the timeout interval will be infinite; otherwise, the timeout interval will be the twos complement of the number sent in centiseconds (e.g., $FE will yield a 20-millisecond timeout).

The sequence for transferring data from the 68xxx to the HP-HIL follows:

1. Disable auto-polling by clearing bit 0 of the `LPCTRL` register.
2. Enable error reporting, if desired.
3. Wait for bit 2 (loop busy) of register 2 to clear.
4. Send the command `$E0` to point the data input pointer to the data buffer.
5. Send three bytes of data; the data input pointer automatically increments.
6. Send the trigger command `$C5` to cause data to be transferred to the loop controller chip and initiate the timeout.
7. If data from the loop is expected, wait for a Status 5 interrupt indicating data from the loop is available or the loop timed out.
8. Do whatever.
9. Re-enable auto-polling.

If the 68xxx-to-loop data transfers complete within 20 milliseconds, no polls will be missed. If not, the next poll will be delayed and may cause the loss of a keystroke or some other data supplied by an unbuffered peripheral.

The New Keymap

HP-HIL 46020A keyboards return both up-stroke and down-stroke keycodes for up to 128 keys. As mentioned before, to minimize changes in the operating systems and boot ROM, the keyboard controller has the capability of mapping the HP-HIL keycodes into the keycodes and protocol expected by the series 200 operating systems. At power up, the default is to map all keyboards on the loop through this protocol conversion algorithm. This protocol conversion, or keyboard cooking, can be selectively or collectively enabled or disabled for any or all keyboards on the loop. The functions provided by the protocol conversion are:

- Map the HP-HIL down-stroke keycodes into series 200 standard keycodes.
- Block all HP-HIL up-stroke keycodes (except for a few special keys).
- Maintain the status of the `Shift` and `Ctrl` keys and return the state of these keys with the mapped keycodes.
- Implement the auto-repeat function with programmable delay and repeat rate.
- Implement the level 7 interrupt reset function (`Shift`-`Break`).

In the table below, the keycodes returned for a mapped keyboard are listed in the two leftmost columns in both hexadecimal and decimal. The next column indicates if the key exists only on the 46020A keyboard ("H", short for "HP-HIL"), only on the old style series 200 keyboard ("I", short for "internal"), or both ("HI"). The next column (IC) is the 46020A keycode returned for keyboards in raw mode (hexadecimal). The next column is the keycap label; "np" means "numeric pad." For a raw mode keyboard, the keycode listed is for the down-stroke, the up-stroke is the same keycode with the least significant bit set. Hence, down-strokes are even and up-strokes are odd.

HEX	DEC	HI	IC	KEY-LABEL	HEX	DEC	HI	IC	KEY-LABEL
00	0			UNUSED	40	64	HI	20	np `1`
01	1	H	7E	`~` `‘`	41	65	HI	24	np `2`
02	2	H	CA	`|` `\`	42	66	HI	28	np `3`
03	3	H	3E	`esc` / `del`	43	67	HI	2E	np `-`
04	4			UNUSED	44	68	HI	10	np `4`
05	5	H	0E	`BREAK` / `RESET`	45	69	HI	14	np `5`
06	6	H	9C	`STOP`	46	70	HI	18	np `6`
07	7	H	EA	`SELECT`	47	71	HI	2A	np `*`
08	8	H	1E	np `ENTER`	48	72	HI	1A	np `7`
09	9	H	4C	np `TAB`	49	73	HI	12	np `8`
0A	10	H	4A	np `K0` (blank1)	4A	74	HI	16	np `9`
0B	11	H	42	np `K1` (blank2)	4B	75	HI	22	np `/`
0C	12	H	46	np `K2` (blank3)	4C	76	I		np `E`
0D	13	H	4E	np `K3` (blank4)	4D	77	I		np `(`
0E	14	H	DC	`↖`	4E	78	I		np `)`

HEX	DEC	HI	IC	KEY-LABEL
0F	15	H	DE	PREV
10	16	H	EE	NEXT
11	17	H	9E	ENTER / PRINT
12	18	H	06	EXTEND (LEFT)
13	19	H	04	EXTEND (RIGHT)
14	20	H	A0	SYSTEM / USER
15	21	H	90	MENU
16	22	H	AC	CLR LINE
17	23	H	AE	CLR DISP
18	24	HI	5E	CAPS LOCK
19	25	HI	6E	TAB
1A	26	I		K0
1B	27	HI	98	K1 (f1)
1C	28	HI	96	K2 (f2)
1D	29	HI	A2	K5 (f5)
1E	30	HI	A4	K6 (f6)
1F	31	HI	A6	K7 (f7)
20	32	HI	94	K3 (f3)
21	33	HI	92	K4 (f4)
22	34	HI	FA	↓
23	35	HI	FC	↑
24	36	HI	A8	K8 (f8)
25	37	I		K9
26	38	HI	F8	←
27	39	HI	FE	→
28	40	HI	BC	INSERT LINE
29	41	HI	BE	DELETE LINE
2A	42	I		RECALL
2B	43	HI	CC	INSERT CHAR
2C	44	HI	CE	DELETE CHAR
2D	45	I		CLEAR→END
2E	46	HI	BA	BACK SPACE
2F	47	I		RUN
30	48	I		EDIT / DISP FCTNS
31	49	I		ALPHA / DUMP ALPHA
32	50	I		GRAPHICS / DUMP GRAPH

HEX	DEC	HI	IC	KEY-LABEL
4F	79	I		np ^
50	80	HI	7C	1
51	81	HI	7A	2
52	82	HI	78	3
53	83	HI	76	4
54	84	HI	74	5
55	85	HI	72	6
56	86	HI	70	7
57	87	HI	B0	8
58	88	HI	B2	9
59	89	HI	B4	0
5A	90	HI	B6	-
5B	91	HI	B8	=
5C	92	HI	C6	[
5D	93	HI	C8	]
5E	94	HI	D6	;
5F	95	HI	D8	'
60	96	HI	E2	,
61	97	HI	E4	.
62	98	HI	E6	/
63	99	HI	F2	SPACE
64	100	HI	C2	O
65	101	HI	C4	P
66	102	HI	D2	K
67	103	HI	D4	L
68	104	HI	6C	Q
69	105	HI	6A	W
6A	106	HI	68	E
6B	107	HI	66	R
6C	108	HI	64	T
6D	109	HI	62	Y
6E	110	HI	60	U
6F	111	HI	C0	I
70	112	HI	5A	A
71	113	HI	58	S
72	114	HI	56	D

HEX	DEC	HI	IC	KEY-LABEL
33	51	I		STEP / ANY CHAR
34	52	I		CLEAR LINE / CLR SCR
35	53	I		RESULT / SET TAB
36	54	I		PRT ALL / CLR TAB
37	55	I		CLR I/O / STOP
38	56	I		PAUSE / RESET
39	57	HI	DA	ENTER (RETURN)
3A	58	I		CONTINUE
3B	59	I		EXECUTE
3C	60	HI	2C	np 0
3D	61	HI	48	np .
3E	62	HI	1C	np ,
3F	63	HI	26	np +

HEX	DEC	HI	IC	KEY-LABEL
73	115	HI	54	F
74	116	HI	52	G
75	117	HI	50	H
76	118	HI	D0	J
77	119	HI	E0	M
78	120	HI	38	Z
79	121	HI	36	X
7A	22	HI	34	C
7B	123	HI	32	V
7C	124	HI	30	B
7D	125	HI	F0	N

Keycodes with the most significant bit set were not used in the previous Series 200 implementation; some of these codes have now been defined for special keys and keys unique to the HP-HIL keyboard. These new keycodes are listed below:

HEX	DEC	HI	IC	KEY-LABEL
–	–	H	0C	CTRL
–	–	H	0A	SHIFT (RIGHT)
–	–	H	08	SHIFT (LEFT)
92	146	H	07	EXTEND CHAR (RIGHT) UP
93	147	H	09	EXTEND CHAR (LEFT) UP

The following keys do not presently exist on any 46020A keyboard but do have matrix locations defined.

HEX	DEC	HI	IC	KEY-LABEL
A9	169	H	5C	NOT NAMED
AA	170	H	E8	NOT NAMED
AB	171	H	EC	NOT NAMED
B0	176	H	80	BUTTON 0
B1	177	H	82	BUTTON 1
B2	178	H	84	BUTTON 2
B3	179	H	86	BUTTON 3
B4	180	H	88	BUTTON 4
B5	181	H	8A	BUTTON 5
B6	182	H	8C	BUTTON 6
B7	183	H	8E	(RESERVED)

The raw mode `BUTTON` codes are the same as the codes returned for buttons on a locater device such as the mouse.

Displays 9

Introduction

The displays are a much simpler subject than the keyboard, but the Model 216, Model 217, Model 226, Model 236 and Model 237 displays are all slightly different. This chapter covers all five displays. References to "the display" should be taken to mean all the versions.

The display is a magnetic-deflection raster device which appears in the 68xxx's address space as a memory-mapped I/O device consisting of dual-port memories and some control registers. The hardware is relatively simple and conceptually quite similar across the family.

There are two main subdivisions of displays in the Series 200 family of computers: the "alpha" type and the "bit-mapped" type. All Series 200 displays *except* the Model 237 display are "alpha" displays; that is, they have independent alpha memory and graphics memory areas which store the respective data to be displayed. Alpha displays can present the alpha (text) image by itself, the graphics (picture) image by itself, both together, or neither. A bit-mapped display has *only* a graphics raster, and all textual information is drawn onto the graphics screen. This means, obviously, that you can't "turn off" the textual information or the graphics information independently.

First, we'll talk about the alpha displays—those capable of displaying two independent rasters.

"Alpha" Displays

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 68xxx bus and instruction set allow use of 6800 family devices. System software must correctly set up the control registers of the 6845.

For the remainder of this chapter, two different character sets will be referred to:

- The "old" character set. This is the character set used by the Models 216, 220 226, and 236. It is called "old" because it is the character set used on early Series 200 machines. This character set consists of the standard ASCII characters for positions 0–127, and a mixture of Roman Extension and Katakana characters for positions 128–255.
- The "Roman 8" character set. This character set is used by the Models 217 (except when shipped to Japan) and 237. This character set also consists of ASCII characters in positions 0–127. However, the upper half of the character set—positions 128–255—are yet another (HP-standard) Roman Extension set. Throughout this chapter, we'll refer to characters 128–255 in the Roman 8 character set as "the upper half of the Roman 8 character set."

Display Hardware Capabilities

Model 216 Alpha

Dimensions Screen size is 210 mm (9″) diagonal, with a useable area of about 168 mm×126 mm. Up to 25 lines of 80 characters can be displayed; Pascal uses the bottom line for a typeahead buffer. See Model 236 discussion.

Character set Same as that of the Model 236 ("old" character set). The character cell is 10 dots wide (there are 800 horizontal alpha dots) by 12 dots high. Characters are displayed in a 7×8 matrix within this character cell, so there are three blank dot columns between characters. A typical character ("A") is 1.5 mm wide by 3.4 mm high, although more dots may be used for descenders.

Attributes The display has no display enhancements unless you have the 09816-66582 board. This board allows all display enhancements, and can be detected by software by attempting to access the highlight byte (the even-numbered byte) of any of the alpha memory. If you get a DTACK (bus) error, you do not have the enhancements board; if you successfully read or write, you do.

The presence of display enhancements is also determined by the Boot ROM, and is "remembered" in bit 2 of `SYSFLAG`. `SYSFLAG` is exported from `SYSGLOBALS`.

The bottom *n* lines can be set up to display in inverse video; this can be used to visually distinguish the soft key area from the rest of the screen. In the soft key area, graphic dots are inverted.

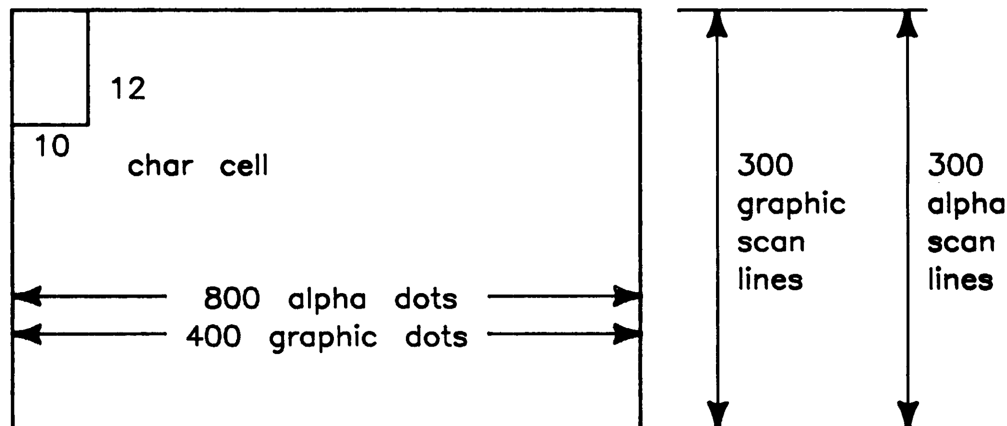
Enabling The alpha display is turned on or off by writing a byte to the 6845 display controller.

Model 216 Graphics

Graphics is an optional feature in the Model 216, 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 0.4 mm per dot), with alpha dots half as wide as graphic dots. Graphic dots are exclusive-ored 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.



Model 216 CRT Viewing Area

Model 217 Alpha

Dimensions Screen size 322 mm (14″) diagonal with useable area about 230 mm×175 mm. 25 lines of 80 characters can be displayed at a time, although the Pascal system uses the bottom line for the typeahead buffer so normally, 24 user lines are available. This typeahead buffer can be moved by software operations.

Character set The Model 217 has can support both the old and the Roman 8 character sets. Roman 8 is the one selected by the factory during manufacturing.

There is a hardware switch which allows you to use either the old set or the Roman 8 set. The switch is hardware-settable, but it is software readable. To determine the setting of the switch from software, look at bit 5 of the memory location `$51FFFE`; 0 = old character set, 1 = Roman 8.

The cell for alpha characters is 9 dots wide by 15 dots high, and characters are displayed in a 7×9 matrix within this character cell. Usually the two extra dot columns are zero. A typical character ("A") is 2.1 mm wide by 4.0 mm high, although more dots may be used for descenders.

Attributes Characters can be displayed in normal, half-bright, inverse video and blinking. Any combination of these attributes can be employed.

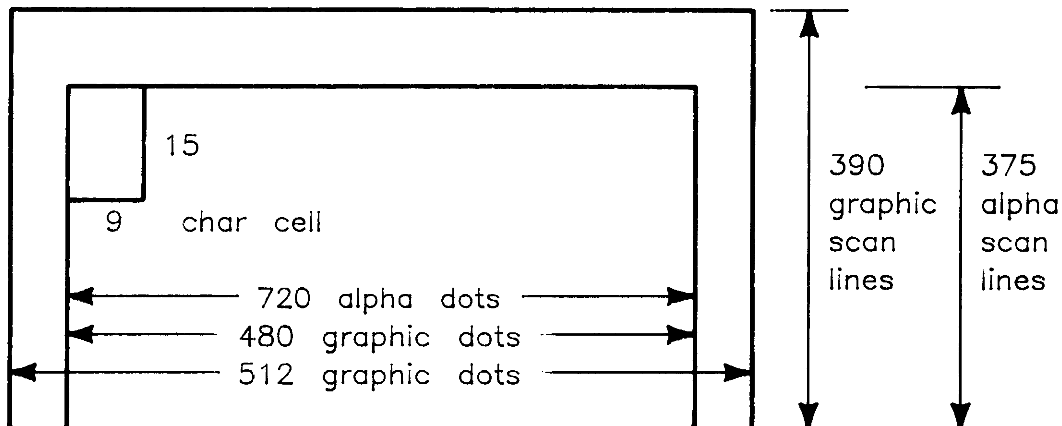
Enabling The alpha display is turned on or off by writing a byte to the 6845 controller.

Model 217 Graphics

Dimensions The 98204B board gives a graphics raster of 512 dots wide and 390 dots high. The graphics screen extends outside the alpha screen by 16 graphics dots on each side, and extends 15 scan lines vertically above the alpha display. Alpha and graphics dots do not overlay; one graphic dot equals 3/2 of an alpha dot.

What happens when graphic and alpha rasters are both displayed? Graphic pixels will invert any coincident *full* bright pixel in the alpha display. Graphic pixels will overwrite (in *full* bright) any coincident *half* bright pixel in the alpha display.

Enabling Graphics display is turned on if its memory is accessed with address line A15 low, and turned off if A15 is high.



Model 217 CRT Viewing Area

Model 220 Alpha

The Model 220 computer comes with an option of either a 9-inch (210 mm) or a 12-inch (292 mm) Nippon Electric Company (NEC) monitor. This affords two different *physical* sizes, but alpha and graphics resolutions are identical on the two monitors.

Since two sizes are available, both will be given in this discussion: the 9-inch values first, then the 12-inch values in brackets.

Dimensions Screen size 210 mm (9″) [292 mm (12″)] diagonal with useable area about 168 mm×126 mm [210 mm×160 mm]. 25 lines of 80 characters can be displayed at a time, although the Pascal system uses the bottom line for the typeahead buffer so normally, 24 user lines are available. This typeahead buffer can be moved by software operations.

Character set The alpha resolution can be either of two different values, independent of the physical size of the monitors. The resolution is determined by the version of the display hardware board: 98204A vs. 98204B. With the 98204A board, the alpha is identical to the Model 216 alpha, and with the 98204B board, it is identical to the Model 217 alpha.

The cell for alpha characters is 10×12, with most characters fitting in a 7×8 matrix within this cell. Characters are displayed so that there are three blank dot columns between characters which are adjacent in the horizontal direction. A typical character ("A") is 1.5 mm wide by 3.4 mm high [1.8 mm wide by 4.3 mm high], although more dots may be used for descenders.

Attributes Characters can be displayed in normal, half-bright, inverse video and blinking. Any combination of these attributes can be employed.

Enabling The alpha display is turned on or off by writing a byte to the 6845 controller.

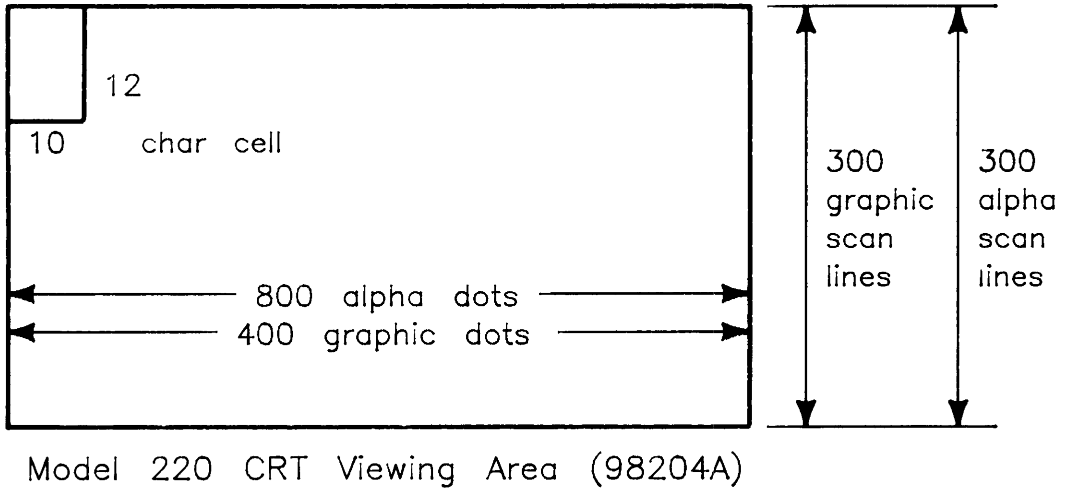
Model 220 Graphics

Dimensions With the 98204A board, you get a graphics resolution of 400 dots wide and 300 dots high (the same as the Models 216 and 226). The graphics screen is exactly the same size as the alpha screen. Alpha and graphics dots do not overlay; one graphic dot equals 2 alpha dots.

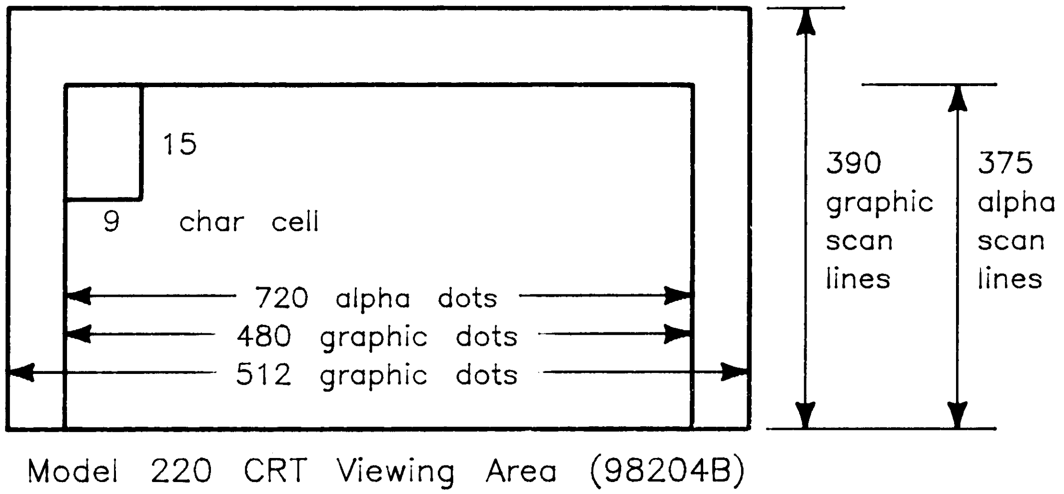
With the 98204B board, you get a graphics resolution of 512 dots wide by 390 dots high (the same as the Models 217, 236A, and 236C). The graphics screen is 512 dots wide and 390 dots high. The graphics screen extends outside the alpha screen by 16 graphics dots on each side, and extends 15 scan lines vertically above the alpha display. Alpha and graphics dots do not overlay; one graphic dot equals 3/2 of an alpha dot.

What happens when graphic and alpha rasters are both displayed? Graphic pixels will invert any coincident *full* bright pixel in the alpha display. Graphic pixels will overwrite (in *full* bright) any coincident *half* bright pixel in the alpha display.

Enabling Graphics display is turned on if its memory is accessed with address line A15 low, and turned off if A15 is high.



Model 220 CRT Viewing Area (98204A)



Model 220 CRT Viewing Area (98204B)

Model 226 Alpha

Dimensions Screen size is 177 mm (7″) diagonal, with useable viewing area about 130 mm×100 mm. Up to 25 lines of 50 characters can be displayed, although Pascal uses the bottom line for a typeahead buffer (see Model 236 discussion).

Character set Same as that of the Model 236 ("old" character set). The character cell is 8 dots wide by 12 dots high, and characters are typically displayed in a 5×7 matrix within the character cell. A typical character ("A") is 1.6 mm wide by 2.3 mm high, although more dots may be used for descenders.

Attributes The Model 226 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-ored 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.

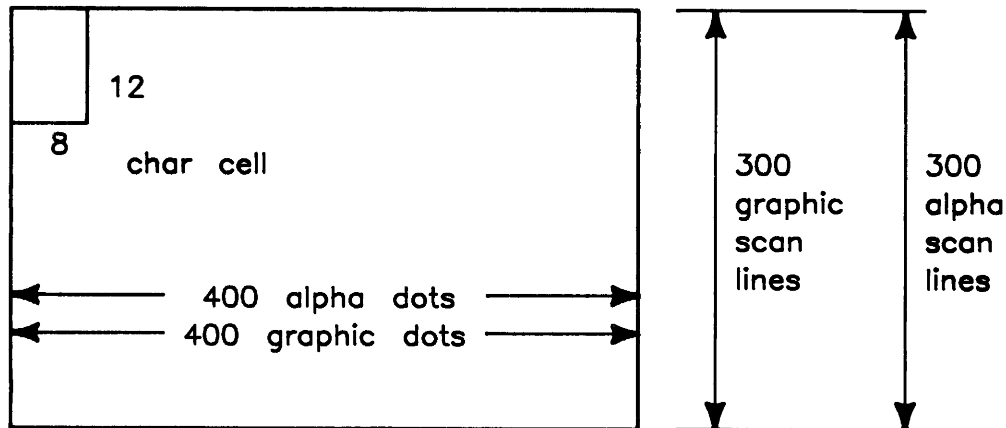
Enabling The alpha display is turned on or off by writing a byte to the 6845 controller (see *Registers 12 and 13* section, later in this chapter).

Model 226 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 0.3 mm per dot).

Normally graphic and alpha dots are exclusive-ored together, that is, an alpha "on" dot will invert a graphic dot and vice-versa. In the "softkey area" at the bottom of the screen, graphic dots are full bright over the half-bright background (inclusive-ored 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.



Model 226 CRT Viewing Area

Model 236A Alpha

Dimensions Screen size is 292 mm (12″) diagonal with useable area about 210 mm×160 mm. 25 lines of 80 characters can be displayed at a time, although the Pascal system uses the bottom line for the typeahead buffer, so normally 24 user lines are available. This typeahead buffer can be moved by software operations.

Character set The "old" character set. The character cell is 9 dots wide by 15 dots high. Characters are displayed in a 7×9 matrix within this character cell, so there are 2 blank dot columns between characters. A typical character ("A") is 1.9 mm wide by 3.7 mm high, although more dots may be used for descenders.

Attributes Characters can be displayed in normal, inverse video, underlined, blinking, and half-bright. Any combination of these attributes can be employed.

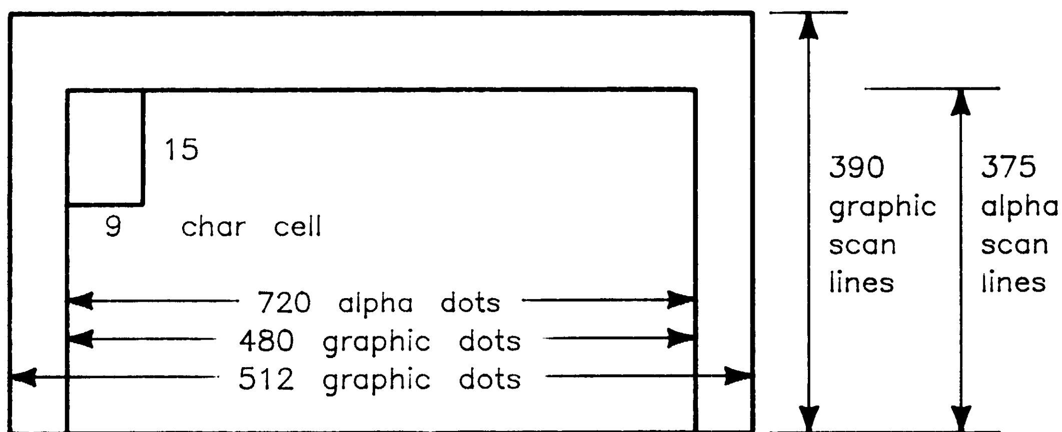
Enabling The alpha display is turned on or off by writing a byte to the 6845 controller.

Model 236A Graphics

Dimensions The graphics raster is 512 dots wide and 390 dots high. The graphics screen extends outside the alpha screen by 16 graphics dots on each side, and extends 15 scan lines vertically above the alpha display. Alpha and graphics dots do not overlay; one graphic dot equals 3/2 of an alpha dot.

What happens when graphic and alpha rasters are both displayed? Graphic "on" pixels will invert any coincident *full* bright pixel in the alpha display. Graphic "on" pixels will overwrite (in *full* bright) any coincident *half* bright pixel in the alpha display.

Enabling Graphics display is turned on if its memory is accessed with address line A15 low, and turned off if A15 is high.



Model 236A CRT Viewing Area

Model 236C Alpha

Dimensions Screen size 292 mm (12″) diagonal with useable area about 210 mm×160 mm. 25 lines of 80 characters can be displayed at a time, although the Pascal system uses the bottom line for the typeahead buffer so normally, 24 user lines are available. This typeahead buffer can be moved by software operations.

Character set Same as that of the Model 236 ("old" character set). The character cell is 9 dots wide by 15 dots high. Characters are displayed in a 7×9 matrix within this character cell, so there are 2 blank dot columns between characters. A typical character ("A") is 1.9 mm wide by 3.7 mm high, although more dots may be used for descenders.

Attributes Characters can be displayed in normal, inverse video, underlined, and blinking. The half-bright bit of the highlight byte is ignored. Any combination of these attributes can be employed. Also, any of the eight "cardinal" colors[1] can be selected for text. In any of these colors, any of the highlights may be used.

Enabling The alpha display is turned on or off by writing a byte to the 6845 controller.

Model 236C Graphics

Dimensions The graphics raster is 512 dots wide and 390 dots high. The graphics screen extends outside the alpha screen by 16 graphics dots on each side, and extends 15 scan lines vertically above the alpha display. Alpha and graphics dots do not overlay; one graphic dot equals 3/2 of an alpha dot.

The graphics display is a color-mapped display; that is, the values stored in graphics screen memory are pointers or indices into the color map. Each pixel is the least significant 4 bits within a byte address. The color map is an area of memory starting at `$51F800` and ending at `$51F81F`. It consists of 16 16-bit integers which are broken up as follows:

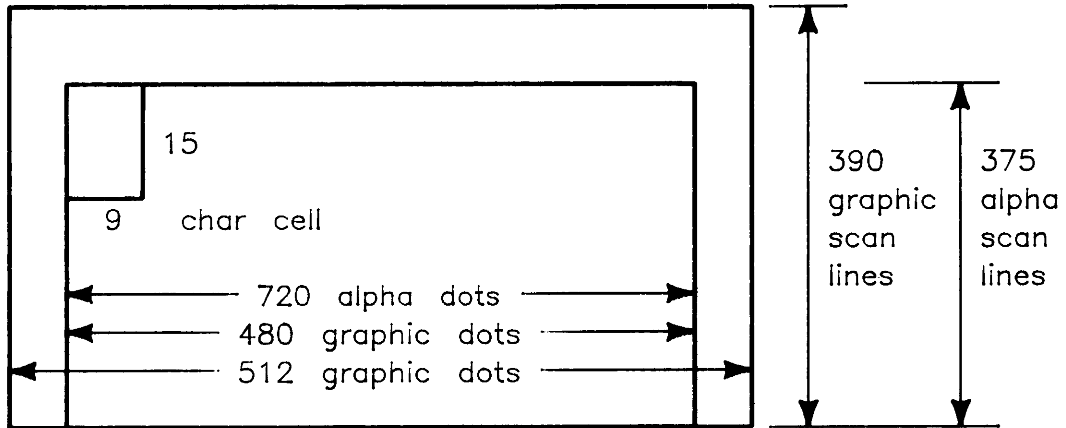
15 12	11 8	7 4	3 0
(unused)	Red	Green	Blue

The Red, Green, and Blue values are the complements of the 4-bit values of the intensity of the respective color. For example, the values in a color map entry specifying magenta would be *xxxx* 0000 1111 0000, where *xxxx* are "don't care" bits. Magenta is composed of full-intensity red and blue, with no green. Note that the values in the color map entry are the *complements* of full-intensity red and blue, with zero-intensity green.

What happens when graphic and alpha rasters are both displayed? The graphics will appear "behind" the text. No pixel inversion will take place, as it does on the Model 236A; if an alpha pixel and a graphics pixel are vying for the same position, the alpha pixel wins.

Enabling Graphics display is turned on if its a 1 is written to the least significant bit of `$51FFFC`, and turned off if a 0 is written there.

[1] A "cardinal" color is one of the eight colors which can be formed by mixing any combination of the three primary additive colors (red, green, and blue) at zero- or full intensity. These colors are black, red, green, yellow, blue, magenta, cyan, and white.



Model 236C CRT Viewing Area

Alpha Screen Driver Considerations

The general process by which alpha characters are displayed on an alpha (not bit-mapped) CRT is as follows. There is a dual-port RAM beginning at `$512000` in the 68xxx's address space. The 68xxx 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 Models 217, 236A, and 216 (with display enhancements), the even-numbered byte of each word of alpha RAM controls the highlight attributes (inverse video, blinking, etc.). In the Model 236C, the even-numbered byte also controls the color of that character. The alpha RAM for the Models 217 and 236s is really 2K words or 4K bytes in size. In the Model 226 and the Model 216 (except when the 216 has display enhancements), there is no highlight byte and only the odd bytes are present; the memory is 2K bytes in size, although the address space still extends over 4K bytes. Since a "line" in the Model 226 is fifty characters wide, there is a good deal of extra memory in the alpha RAM on that machine.

A programming note about accessing alpha RAM. In the Model 226 and Model 216 (except when the 216 has display enhancements), a bus error will occur if the 68xxx 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 (i.e., will work on any flavor of alpha-display 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 68xxx bus, operations such as scrolling and window management are easily implemented in software. To scroll a region of the CRT, the 68xxx simply shifts the whole region "up" or "down" by copying the data and highlight bytes. This is a very fast operation if done with move-multiple (`MOVEM`) instructions.

"Bit-Mapped" Displays

Whereas all the Series 200 computers which support graphics have bit-mapped *graphics*, they have *non*-bit-mapped alpha. That is, an alpha character is generated by merely writing one byte—the ASCII value of the desired character—to the appropriate memory location.

The only Series 200 machine whose CRT is *entirely* bit-mapped—both graphics and alpha—is the Model 237. The alpha on this machine is generated by placing the appropriate pixel-by-pixel dot pattern onto the graphics display. Alpha output and graphical output are sent to the same address area; thus, there is no difference between the "alpha" memory and the "graphics" memory—they are one.

This is a good place to introduce the term *frame buffer.* All the other Series 200 graphics displays (graphics *only*) were also frame buffers. On all the Series 200 machines, the total graphics memory is somewhat larger than the amount which is actually displayed, because the the address space of the graphics memory is "square" (i.e., it describes enough pixels for a square display). The *displayed* screens are not square—they are wider than they are tall, and the extra memory is unused. Because the only graphics memory used for anything prior to this point was *displayed* graphics memory, we didn't really need the term "frame buffer;" the terms *graphics display* and *frame buffer* could have been used interchangeably.

In the Model 237, the frame buffer is a 1024×1024 pixels, whereas the graphics display—those pixels which you can actually see—is only 1024×768 pixels. It would seem, from this, that one quarter of the graphics memory is wasted. The other quarter of the frame buffer is not wasted, however. During different parts of CRT initialization, various bit patterns are placed in this invisible area of the screen memory, so subsequent operations need only *copy* data from one place to another, rather than *unpack* the data from the ROM storage area. For example, during the initialization done by module `CRTB`, the character set is unpacked from the "font ROM," an area of address space containing pixel images for each of the 256+128 displayable characters. Then, since the pixel images are formatted for the frame buffer (rather than in ROM "packed" format), high-speed frame-buffer-to-frame-buffer pixel moves can take place for alpha generation[2] by the bit-mapped CRT driver `CRTB`.

Another example of the use of the invisible part of the frame buffer is for polygon dither-fill patterns. During the initialization performed by DGL's `DISPLAY_INIT`, the 17 different dithering patterns are placed into the unseen part of the frame buffer; again, so fast copies can be done.

As with the "alpha" displays, the CRT control signals are generated by a 6845 chip, which is a member of the 6800 family peripheral devices. The 68xxx bus and instruction set allow use of 6800 family devices. System software (`CRTB` and `GLE_ARAS_OUT`) must correctly set up the control registers of the 6845.

[2] Since characters are stored in the frame buffer—read/*write* memory—this means that you can modify the character set after CRT initialization, as long as the character *size* doesn't change. **Note: we do not recommend doing so!**

Display Hardware Capabilities

Model 237 Alpha/Graphics

The Model 237 has a bit-mapped display, and it is currently the only Series 200 computer which has one. Thus, it has no separate alpha and graphics rasters; they are one.

Dimensions Screen size 412 mm diagonal with useable area about 312 mm×234 mm. 48 lines of 128 characters can be displayed at a time, although the Pascal system uses the bottom line for the typeahead buffer, so normally 47 user lines are available. This typeahead buffer can be moved by software operations.

Character set Characters are defined by a font ROM containing the Roman 8 character set plus the upper half of the "old" character set. In the font ROM, these characters in a packed bit-per-pixel format, not the frame buffer byte-per-pixel format.

The character cell is 8 dots wide by 16 dots high. Characters are displayed in a 6×10 matrix within this character cell, so there are two blank dot columns between characters. A typical character ("A") is 1.8 mm wide by 3.0 mm high, although more dots may be used for descenders.

Attributes Characters can be displayed in normal, underlined, and inverse video modes. Any combination of these attributes can be employed, although none of these attributes can be obtained *through the hardware;* they are all software functions. Blinking and/or half-bright text are not possible.

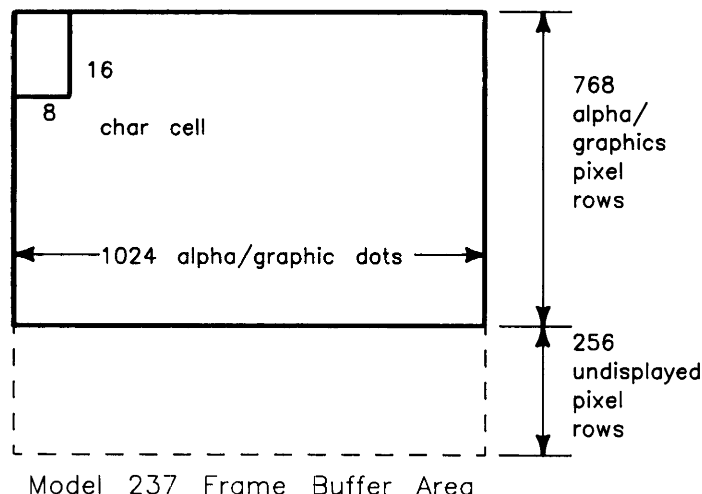
Enabling The alpha/graphics display is controlled by the 6845 at addresses $566001 (register select) and $566003 (data).

Model 237 Graphics

There is no way to do an alpha clear without a graphics clear or vice versa.

Dimensions The displayable alpha/graphics raster is 1024 dots wide and 768 dots high. As mentioned before, the alpha screen *is* the graphics screen.

Enabling The alpha/graphics display is controlled by the 6845. It is turned on by writing a 1 to bit 5 of the 6845 register R12. A zero will turn it off.



Model 237 Frame Buffer Area

Alpha Screen Driver Considerations

Alpha Displays

In the Models 216 (with display-enhancement board), 217 and 236 (these have "alpha" displays), alpha characters can be displayed using any combination of these attributes: inverse video, blinking, underlined, half-bright[3]. The display attributes of a character (which is in alpha RAM but looks to the 68xxx like an odd-numbered byte of its address space) are stored in the immediately preceding even-addressed byte of 68xxx memory (alpha RAM).

The attributes are governed by the low-order 4 bits of the attribute byte as follows:

Inverse video	iff Bit 0 = 1	
Blinking	iff Bit 1 = 1	
Underlined	iff Bit 2 = 1	
Half bright	iff Bit 3 = 1	(where the hardware supports it)

For the Model 236C, the color is governed by the next higher three bits of the highlight byte:

Red	iff Bit 4 = 1	(where the hardware supports it)
Green	iff Bit 5 = 1	(where the hardware supports it)
Blue	iff Bit 6 = 1	(where the hardware supports it)

By selecting various combinations of these three bits, you can get text in the following eight colors:

Black	Bit 6 (blue): 0; bit 5 (green): 0; bit 4 (red): 0.
Red	Bit 6 (blue): 0; bit 5 (green): 0; bit 4 (red): 1.
Green	Bit 6 (blue): 0; bit 5 (green): 1; bit 4 (red): 0.
Yellow	Bit 6 (blue): 0; bit 5 (green): 1; bit 4 (red): 1.
Blue	Bit 6 (blue): 1; bit 5 (green): 0; bit 4 (red): 0.
Magenta	Bit 6 (blue): 1; bit 5 (green): 0; bit 4 (red): 1.
Cyan	Bit 6 (blue): 1; bit 5 (green): 1; bit 4 (red): 0.
White	Bit 6 (blue): 1; bit 5 (green): 1; bit 4 (red): 1.

Bit 7 of the highlight byte is unused.

[3] The Model 236C cannot display half-bright characters.

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 68xxx but not read. In the various Series 200 computers, attempting to read them will either produce a bus error, or, if no bus error occurs, the data returned will be garbage.

Writing to a 6845 register is a two-step process. First you must write to address `$510001` (`$566001` for the Model 237), 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` (`$566003` for the Model 237). See the example program below.

The first 10 registers of the 6845 are initialized at boot time with certain 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. *The following three sections, discussing registers 10 through 15, do not apply to the Model 237.*

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 Models 217 and 236 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 Model 216, 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 6 is zero if the cursor should not blink; one if the cursor should blink. Bit 5 is set to zero if the desired blink rate is 1/16 of the vertical frame rate (fast blink) and set to one for a blink every 1/32nd of the vertical frame rate (slow blink). Here is the full table of meanings for bits 6 and 5:

6	5	Meaning
0	0	Non-blinking cursor (on all the time)
0	1	Cursor non-display mode (off all the time)
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 (file "CRSRSIZE.TEXT" on your EXAMPL: disc) to see its effect on the cursor shape. The normal cursor setting for the Pascal cursor is register 10=01001100 and register 11=00001101. Note: *This will work only on non-bit-mapped displays.*

```
$sysprog$
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 Series 200 machines (except the 237), 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 Model 236 uses the `KEYS` and `TEXT` bits to turn on/off either or both sections of the screen:

TEXT	KEYS	Function
0	0	Neither text nor soft key areas are displayed.
1	0	Text area displayed, soft key area off.
0	1	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!

When using this information, don't forget that an eighty-character-wide screen requires different setup values than a fifty-character-wide 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.

Controlling the Model 237 Bit-Mapped Display

Since bit-mapped displays have only one raster—unlike alpha-type displays, which have alpha and graphics rasters—all information displayed is a result of direct bit-moving operations. This is different from the alpha portion of an alpha-type display, where moving one byte of information to the alpha memory results in a complex bit pattern—the character—on the screen.

The Replacement Rule

When moving data to the frame buffer in a bit-mapped display, there is a rule involved which defines the bit replacement in the memory. First, there is the old bit pattern: the one there before the output operation takes place. The "old" bit pattern is called the *destination* pattern, because that is where the new bit pattern is sent. Second, there is the information which you are sending to the frame buffer. This is the *source* bit pattern. And third, there is the bit pattern on the screen *after* the output operation takes place. This is the *resultant* bit pattern.

The rule whereby the resultant bit pattern is defined is called the *replacement rule.* There are sixteen replacement rules, and they are labelled according to the left-to-right, top-to-bottom bit pattern in all the possible truth tables[4]. For example, suppose you want a replacement rule in which the result bits are the source bits *or*ed with the destination bits. The truth table would look like this:

		Destination	
		0	1
Source	0	0	1
	1	1	1

Reading the bits in a left-to-right, top-to-bottom manner, you see that the bits are `0111`, or 7 decimal. Indeed, replacement rule 7 causes the source to be *or* ed with the destination.

The four most commonly-used replacement rules are:

0 Result:=0 (clear bits, regardless of source).

3 Result:=Source (dominant). Default.

7 Result:=Source or Destination (non-dominant).

10 Result:=not Destination (complement).

Note: All of the possible replacement rules do not have intuitive uses; that is, they are not all "useful."

[4] This is the current definition of the replacement rule; there is no guarantee that is will always remain this way.

The other replacement rules follow:

Rule Number		Replacement Rule
0	(0000)	0—clear pixel regardless of Source or Destination
1	(0001)	`Source` *and* `Destination`
2	(0010)	`Source` *and* (*not* `Destination`)
3	(0011)	`Source` (write data as sent)
4	(0100)	*not* `Source` *and* `Destination`
5	(0101)	`Destination` (no change)
6	(0110)	`Source` *exor* `Destination`
7	(0111)	`Source` *or* `Destination`
8	(1000)	`Source` *nor* `Destination`
9	(1001)	`Source` *exnor* `Destination`
10	(1010)	*not* `Destination` (complement pixel presently on screen)
11	(1011)	`Source` *or* (*not* `Destination`)
12	(1100)	*not* `Source`
13	(1101)	*not* `Source` *or* `Destination`
14	(1110)	`Source` *nand* `Destination`
15	(1111)	1 (set pixel regardless of Source or Destination)

The replacement rule is specified by writing a byte into location $4009 of the Display Control Block[5].

Using the Line-Mover

The hardware for the Model 237 display provides the capability to move pixels in a given scan line to another line. Up to a full 1024-pixel line may be moved. This is not a full block mover, but it provides a significant performance improvements for scrolling and window-moving operations, when compared to simply reading and writing individual pixels.

There are three registers used to control the line-move operation:

- Replacement Rule,
- Window Width, and
- Status Register.

[5] The Display Control Block is a 64K byte area which is just a block of addresses starting at $560000. Part of the control block is ROM (odd bytes only), and part is registers. Through this memory-mapped area, control registers, the CRT controller, the color map, and ID/initialization ROM are accessed. The actual address of this area depends on the select code of the 09920-66562 High Resolution Display controller board.

Bit 7 of the replacement rule register selects the moving mode. When this bit is 0, the system is in pixel mode, the system is in pixel mode, and pixels may be read and written normally, treating each pixel as one byte of memory. When bit 7 is 1, the system is in line-move mode, which operates somewhat differently.

A line move is requested by setting the replacement rule and the window width as desired. The window width is set by writing a full word (twos-complement 16-bit integer) to address $4000C of the Display Control Block, and then performing a byte read and a byte write. The read address is used as the address of the first pixel in the line to be moved; the write address provides the location of the first pixel in the line being moved to. Both addresses must be addresses within the frame buffer. Actual addresses corresponding to X,Y locations must precomputed; this is a *line* mover, not a *block* mover. No valid data is actually transferred between the 68xxx and the frame buffer during these operations.

After a line move is requested, the 68xxx must wait for bit 7 of the Status register (at address $4001 in the Display Control Block) to go high, indicating the line mover is ready/completed, before initiating another frame buffer operation. This is not required during a pixel read or write.

NOTE

The replacement rule, chosen by the lower four bits in the replacement rule register, is also used in pixel operations; therefore, it is *always* important to have this register set correctly before performing any operation on the frame buffer.

While the line-mover has been designed for optimum speed in vertical moves (as in scrolling), horizontal moves may also be accomplished. An additional complication must be considered here: while right-to-left moves will always work, hardware limitations require special care to be taken for left-to-right moves. If a line is to be moved more than 31 pixels to the right, *and* the destination overlaps with the source, the line should not be moved directly, but instead should be copied to some off-screen location within the frame buffer, and then moved from there to the destination. The word "overlap," in the context of the previous sentence, means that the source and the destination are on the same scan line and the distance between their start addresses is less than their lengths.

Model 237 Frame Buffer Allocation

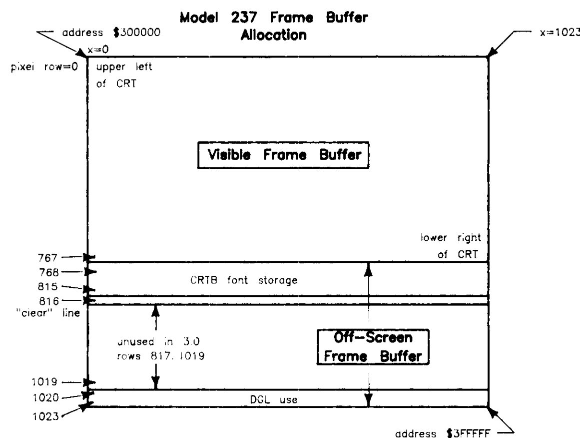
The Model 237 frame buffer has 1024 rows of 1024 pixels, each at 1 byte per pixel. Of these, only 768 rows are displayed on the CRT, leaving 256 rows off the screen. These undisplayed rows can be handled by the line mover in the same way as can displayed pixel rows. However, you must be careful if you choose to store data in the offscreen rows; the `CRTB` driver and DGL both "own" some of these offscreen areas.

`CRTB` uses 48 pixel rows to store the alpha fonts. There are 128 ASCII characters, plus 128 Roman 8 upper-half characters[6], and a set of 128 characters for use with the Japanese language option. This last set is the "old" character set—compatible with Models 216, 226, 236, and the 98204A/B display board fonts. The fonts are arranged in the order indicated above, and within each font, the characters are laid out in index sequence. That is, the first byte of row 768 contains the upper left pixel of the 0th character of the ASCII set (this character is the ASCII null: ␀), and the next byte contains the second-from-left pixel of the top row of that same character, etc. Since each character is 16 pixels high by 8 pixels wide, each character occupies 128 pixels. This yields 8 characters per pixel row, and thus each 128-character font occupies 128/8=16 pixel rows.

The next pixel row after the three fonts (row 816) is reserved for used by the `CRTB` driver as a "clear line," for erasing whole pixel rows at a time, and should not be modified if `CRTB` is installed.

DGL also claims 4 pixel rows. These rows are located at the very bottom of the frame buffer—rows 1020–1023. DGL uses them to do dithered polygon fill. They are set up to the present dither pattern, which is a 4×4 pixel cell. The pattern is replicated horizontally 1024/4=256 times to fill the pixel rows. When a polygon pixel row is to be dither-filled, DGL uses the line mover to move a section of the appropriate dither-pattern line to the visible part of the display, thus performing a fairly high-speed dither fill. If DGL is active and the display has been initialized by `DISPLAY_INIT`, these bottom 4 rows should not be touched. Conversely, doing a `DISPLAY_INIT` of the Model 237 display will change whatever was in these rows.

The other 203 rows (pixel rows 817–1019) are unused by the Pascal Workstation 3.0. This fact does not guarantee that in future revisions, we may not choose to make use of some of this "free" space.



[6] The "Roman 8" upper-half character set is an HP standard extension character set, accessible by using the Extend Char to shift the alpha keys on the 46020A keyboard.

Caveats

There are several important things to realize when dealing with the bit-mapped display, line mover, etc. Since the replacement rule register and the window width registers are not readable, and since the CRT alpha driver may operate in interrupt mode (e.g., to print a character to the type-ahead buffer), a program using the line mover must protect itself against the CRT driver changing the replacement-rule and window-width registers in the ISR. *Before* setting these non-readable registers, set the new value into the appropriate register *copy* variable exported from SYSDEVS, *then* set the register. The CRT driver, if operating under interrupt, will, when through, restore the hardware registers to the values stored in the appropriate register copy variables.

See the listings of the Model 237 CRT driver (CRTB) for proper usage of those registers in an ISR. Note that the copies of the non-readable control registers are saved on the stack, and then these copies are restored and also written to the control registers when use of the line-mover is completed.

NOTE

DGL is not "polite" with the use of these registers. It blithely overwrites with the needed values and *does not* restore them. Thus, upon return from a DGL routine which uses the line mover, do not expect any of the line mover registers to be the same as before the call.

Graphics Screen Driver Considerations

The graphics frame buffer is a much simpler subject than the alpha display. Again it is simply a dual-port memory accessible both by the 68xxx and by the CRT display circuitry.

	Model 216	Model 217	Model 226	Model 236A	Model 236C	Model 237
Width (mm)	168	230	130	210	210	312
Height (mm)	126	175	100	160	160	234
Width (pixels)	400	512	400	512	512	1024
Height (pixels)	300	390	300	390	390	768
Pixels/mm	2.38	2.23	3.08	2.44	2.44	3.28
mm/pixel	0.42	0.45	0.33	0.41	0.41	0.30
Start address	$530001	$530000	$530001	$530000	$520000	$300000
Last pixel address	$537531	$536180	$537531	$536180	$550BFF	$3BFFFE
Ending address	$537FFF	$537FFF	$537FFF	$537FFF	$550BFF	$3FFFFE
Addressed Memory	$7FFF	$7FFF	$7FFF	$7FFF	$30C00	$FFFFF
Actual Memory	$3FFF	$7FFF	$3FFF	$7FFF	$18600	$20000
Visible memory	$3A98	$6180	$3A98	$6180	$30C00	$18000
Address layout	[7]	[8]	[7]	[8]	[9]	[10]

The Model 220 has two different sizes of monitors, as well as two different alpha and graphics resolutions, depending upon whether the 98204A or 98204B display board is installed. When the 98204A board is installed, the alpha and graphics resolutions are the same as those of the Model 216; when the 98204B board is installed, the alpha and graphics resolutions are the same as those of the Model 236A.

The graphics memory allocation for the some of these frame buffers is not straightforward. For example, both the Models 216 and 226 have a graphics resolution of 400×300=120 000 pixels. This requires 15 000 bytes of memory. However, since only odd bytes have RAM loaded—$530001, $530003, $530005, et cetera—and since the ending address minus the starting address is 32K, you would think that the computer has 32K of memory, of which it is wasting 16K. This is not the case. It has only 16K of actual graphics memory; it's just the addressing which is unusual and "wasted."

Similarly, the Model 236C uses one byte per pixel in addressing the frame buffer. Of each byte, only four bits are used (loaded with RAM), but, again, it is not wasting 50% of the frame buffer memory. It is efficient in the actual use of the memory. Likewise, the Model 237, which addresses at one byte per pixel, of which only *one* bit (the least significant) per byte is used.

[7] The addressing mode of this graphics frame buffer is one bit per pixel, odd bytes only. Even *byte* addresses give DTACK error, even *word* addresses are okay.
[8] The addressing mode of this graphics frame buffer is one bit per pixel, all bits utilized (not restricted to odd bytes only).
[9] The addressing mode of this graphics frame buffer is one byte per pixel, where only the 4 least significant bits are stored. The pixel value is an index into the 16-entry color map.
[10] The addressing mode of this graphics frame buffer is one byte per pixel, where only the least significant bit is stored.

Model 236 graphic memory is accessed as word-wide, while the Model 226 and Model 216 are accessed as odd bytes.

As this table indicates, the Models 236 and 237 have much more graphics memory than the other family members. The table also shows that many machines have 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. But watch out: the alpha font and possibly the dither patterns are in extra frame buffer memory on the Model 237 bit-mapped display.

The display operation of frame buffers which do *not* use one byte per pixel is very simple. Bytes are accessed sequentially (skipping even bytes on the Models 216 and 226), 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 Model 236A uses 64 bytes per raster line; the Model 226 and Model 216 use 50 bytes per line but 100 addresses per line.

When accessing graphics memory in a non-bit-mapped, non-Model 36C display, 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 `$538`*xxx* range will turn *off* the display; any access in the `$530`*xxx* 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
CHR(1)	Homes cursor to upper left corner.
CHR(7)	Beeps.
CHR(8)	Moves the cursor left one place if possible.
CHR(9)	Clears from present cursor position to end of line.
CHR(10)	Moves the cursor down one place if possible.
CHR(11)	Clears from present cursor position to end of screen.
CHR(12)	Homes cursor and clears screen.
CHR(13)	Moves cursor to left end of line.
CHR(28)	Moves the cursor right one place if possible.
CHR(31)	Moves the cursor up one place if possible.
CHR(128)..CHR(143)	Cause display highlighting on machines which support it.
CHR(136)..CHR(143)	Cause color highlighting on Model 236C. Although it looks like CHR(136) through CHR(143) have already been used for highlighting, there is really no ambiguity. These eight values, which one would expect to specify halfbright, are not needed on the Model 236C, which does not support halfbright. Therefore, these values are available on the 236C to specify color.

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, 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 or changed by specifying another highlight combination.

The significance of the values is straightforward; as explained under the subheading *Alpha Screen Driver Considerations,* 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.

Scrolling

Pascal treats the CRT as having 24 lines of 50 or 80 characters (for the non-bit-mapped displays) or 47 lines of 128 characters (for the bit-mapped display). Output is always written to the current cursor position, and then the cursor is advance 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 (`CHR(10)`) is written in the last line, and will downscroll if a US ("unit separator," or `CHR(31)`) character is written in the top line.

It is possible to change the dimensions of the CRT scrolling area by modifying some variables exported from `SYSDEVS`. This is done by changing `SYSCOM^.CRTINFO.HEIGHT` and `SYSCOM^.CRTINFO.MAXY` (the latter is always one less than the former). `MAXY` is the number of the line from which scrolling occurs; i.e., the screen scrolls up whenever a `WRITELN` is done to line `SYSCOM^.CRTINFO.MAXY`.

Lower-Level Access to the CRT

At this point we begin to discuss features which can be gotten at through the `SYSDEVS` module. 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 four ways to move the cursor around:

- Write to the 6845 CRT controller directly. This is not a good idea, because then the Pascal system software will 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 (actually a FIB) and two integers:

  `fgotoxy(f,0,0)` move the cursor
  and
  `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`. Use the file `OUTPUT` for `F`.
- Assign values to `XPOS` and `YPOS`, exported from `SYSDEVS`, and then execute `CALL (UPDATECURSORHOOK);`.

Interrogating the Dimensions of the CRT

The variables `SYSCOM^.CRTINFO.WIDTH` and `SYSCOM^.CRTINFO.HEIGHT`, exported from `SYSDEVS`, give this information.

Turning the Screens On and Off

The exported variables `ALPHASTATE` and `GRAPHICSTATE` are true when the respective screens are visible. These are always true on a bit-mapped display.

To change the state of either screen, use the system programming language extension `CALL` on the exported hook procedures `TOGGLEALPHAHOOK` and `TOGGLEGRAPHICSHOOK`:

```
call(togglealphahook);
```

Note: this makes sense only on non-bit-mapped displays, as the operating system never "turns off" the bit-mapped display.

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(dumpgraphicshook);
```

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 or to a file. After implementing the procedure which will dump the output, assign it to the appropriate hook procedure variable. These hook procedures get called automatically when the operator presses a key which causes alpha or graphics to be dumped.

Internal Disc Drives 10

Floppy Control Board

This chapter discusses the interface to the mini-floppy drive which is built into the Series 200 Model 226, 236A, and 236C computers. The sub-assemblies which comprise the built in mass storage include:

1. 9131G double-sided, double-density floppy drive
2. 09826-66561 or 09826-66562 floppy control board

The 09826-66561 control board was used in early Model 226s, 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 Models 226 and 236, 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 a 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 Model 226/236 bus with up to two 5.25 inch double-density, double-sided drives. The drive has 270K bytes of capacity when formatted to the HP Logical Information Format (LIF) standard. The actual data transfer rate is 16K bytes 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)×(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` (4456448 decimal) to `$44FFFF` (45219783 decimal). The space is decoded by the Model 226/236 motherboard to a single line called chip select floppy (`CSF'`).

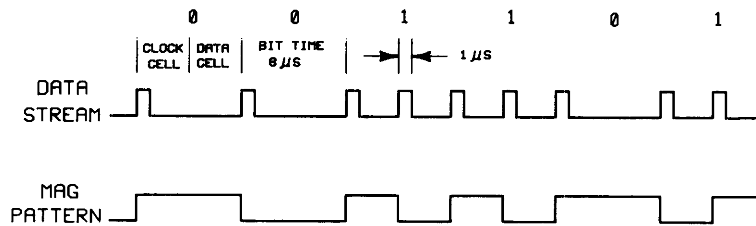
Data transfers

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.

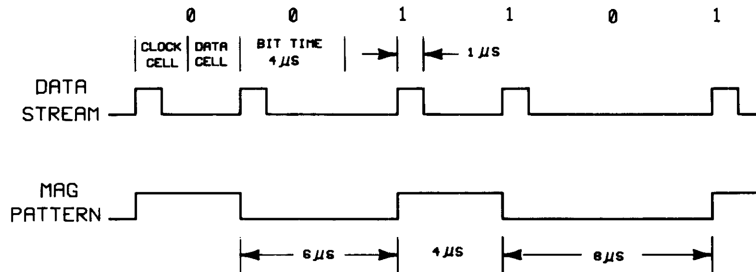
Data Encoding Format

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 μsec. Only clock pulses are present if the bit pattern is all zeros. A one is formed by 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 μsec long. Every clock cell has a pulse. Only data cells which are ones have pulses.

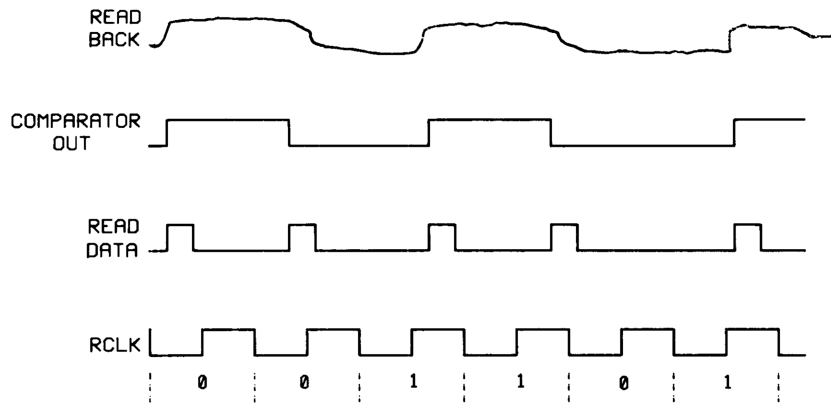


FM DATA ENCODING

MFM encoding is based on a series of clock pulses which occur 4 μsec 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 in which they start.



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-	44E000	4513792	READ-WRITE	1 μs
BOARD RAM	44E1FF	4514303		
COMMAND	44C000	4505600	WRITE	1 μs
STATUS	44C000	4505600	READ	1 μs
TRACK	44C002	4505602	READ-WRITE	1 μs
SECTOR	44C004	4505604	READ-WRITE	1 μs
DATA	44C006	4505606	READ-WRITE	1 μs

A detailed description of the controller board registers follows:

Extended Command

The extended command register (at 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) — Used to select a floppy drive. ADDRESS 1 is not used in the Models 226 and 236. Setting ADDRESS 0 to 0 selects drive 0. Setting ADDRESS 0 to 1 selects drive 1 (left-hand drive in Model 236).

Extended Status

The extended status (XSTAT) register (at 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) — 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) — 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 (CLRXSTAT) special function address.

MARGIN ERROR (bit 2) — 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 CLRXSTAT special function address.

INTERRUPT (bit 3) — This 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 (CLRXSTAT)

The extended status clear register (at address $445400) is a write-only register which, when written to, will clear the extended status register.

Command

The command register (at 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 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 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 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 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 is 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 Model 226/236 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`) 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 Model 226/236 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 milliseconds 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 Model 226/236 systems. The stepping rates for each combination of R1 and R0 with the E and V flags cleared are shown in the table below.

Stepping Rates

R1	R0	milliseconds
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 to abort the current command and issue 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 milliseconds 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 milliseconds 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, are 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 Model 226/236A 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`) 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 is 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 milliseconds 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.

The Boot ROMs 11

Introduction

This chapter of the System Designer's Guide deals with the Boot ROMs, and their functions during the booting of an operating system. For information on creating bootable files, which the Boot ROM would load, see the section *Creating a Bootable System*, toward the end of this chapter, and the *Librarian* chapter of the *Pascal 3.0 Workstation System* manual.

The Boot ROM is a Read-Only Memory located at physical address zero in all of Hewlett-Packard's Series 200 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 five versions of the Boot ROM (1.0 for Model 226 only; 2.0 for Models 226 and 236 only; 3.0 for Models 226, 236, 216S, and 220; 3.0L for Model 216A only; and 4.0 for Models 226U (upgraded from a 226), 236AU, 236CU, 220U, 217, and 237). 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 version number displayed on the screen at powerup. The 1.0 and 2.0 Boot ROMs do not display the version number at powerup. (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 3.0 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 ROMs (3.0 and later) were 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 4.0 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, *never* use 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 (which includes select code, bus address, media format, and device type) is one of the responsibilities of the Boot ROM. A section of this chapter is devoted to how this is done, why it is done, and possible values for the default mass storage device specifier.

Many of Hewlett-Packard's Series 200 products contain a software-readable ID PROM (Programmable Read-Only Memory) with the serial number and product number in it. Determining if the ID PROM is present and reading it are explained in this manual.

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 Series 200 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″ flexible disc drive is 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.

In several places in this chapter, byte subfields of one or more bits are referred to. These subfields are represented by the notation $\langle i{:}j \rangle$, where the subfield consists of bits i through j, inclusive. For example, $\langle 3{:}0 \rangle$ denotes the bit 3 through bit 0 of a byte.

Again in this chapter, a number preceded by a dollar sign—for example, `$FFF001`—is a hexadecimal number.

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 executed out of that memory.

Explained first in this section 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. Next, the ROM/EPROM Disc format allowed for a soft system is explained. Finally, booting soft systems from the Shared Resource Manager (SRM) is explained.

NOTE

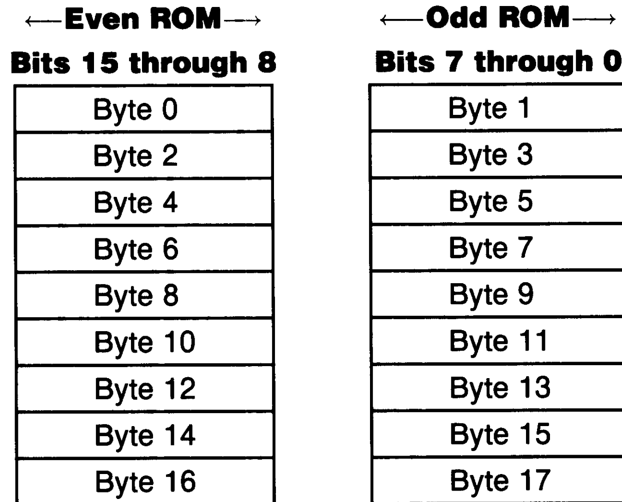
Some versions of the Boot ROM do not support all boot formats. Only the 3.0 and later Boot ROM versions support all the boot formats mentioned here. In the text that follows, watch for notes indicating limitations.

ROM Headers

The Boot ROM will find a ROM (hard) system in ROM space if that ROM system starts with a valid ROM header.

The ROM header is the first 18 bytes found at the beginning of each 16K byte boundary in ROM space. ROM space begins at 256K bytes for 1.0 and 2.0 Boot ROMs, and at 64K bytes for the 3.0, 3.0L and 4.0 Boot ROMs. ROM space ends at 4M bytes. (Note: in the 9837A, the address space from `$300000` to `$3FFFFF` is "stolen" by the frame buffer and future hardware may steal other areas in the ROM space.)

ROM Header (Starting at 16K byte Boundary)



ROM headers are 18 bytes long, unless bytes 14 and 15 both have only bit 6 set in bits ⟨7:3⟩, in which case the header is 16 bytes long. All currently-supported systems have 18-byte headers; 16-byte headers only occurring in the earliest Series 200 machines.

For all discussions that follow: bit 0 is the least significant bit, or "LSB." 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 by HP are: `B` For BASIC `H` For HPL
3		Flag: Flag for type of ROM.
	⟨0:0⟩	1 = System ROM (LSB of the byte)
	⟨1:1⟩	1 = Boot Extension ROM.
	⟨2:2⟩	1 = Language Extension ROM
	⟨3:3⟩	1 = Pseudo-disc follows header in memory (not supported by 1.0 or 2.0 Boot ROMs.) (See *ROM/EPROM Disc* section.)
	⟨4:4⟩	1 = ROM should not be checksummed
	⟨5:7⟩	Reserved for HP, set to 0.

Byte	Bits	Description
4–7		Execution start address of a Boot Extension ROM, relative to the start of the header. If this is not a Boot Extension ROM, then this location can be used for storing a checksum (as long as byte *3* has bit 1=0 and bit 4=0). 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 ⟨5:0⟩ are the part number for the even ROM in a ROM pair (1 is the first valid number, 3 is the next valid number, etc.). Bits ⟨7:6⟩ are the system revision number for the even ROM in a ROM pair (0 is the first valid number). The 3.0 and later Boot ROMs verify that bytes 12 and 13 have consecutive part number values.
13		Odd Part ROM Number and System Rev: Bits ⟨5:0⟩ are the part number for the odd ROM in a ROM pair (2 is the first valid number, 4 is the next valid number, etc.). Bits ⟨7:6⟩ 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 ⟨2:0⟩ are the part revision for the even ROM in a ROM pair (1 is the first valid number). Bits ⟨7:3⟩ are system capability bits. Capability bits are only looked at by the Boot ROM if the system bit, bit 0 of byte 3, is set.
	⟨7:7⟩	Reserved for HP, set to 0.
	⟨6:6⟩	A value of 1 means that the system can handle a 50-character-wide CRT.
	⟨5:4⟩	Reserved for HP, set to 0.
	⟨3:3⟩	A value of 1 means that the system can handle an 80-character-wide CRT.
	⟨2:0⟩	Part revision for the even ROM in a ROM pair.
15		Capability Bits II and Odd Part Rev: Bits ⟨2:0⟩ are the part revision for the odd ROM in a ROM pair (1 is the first valid number). Bits ⟨7:3⟩ are system capability bits:
	⟨7:7⟩	Reserved for HP, set to 0.
	⟨6:6⟩	Reserved for HP, set to 1.
	⟨5:3⟩	Reserved for HP, set to 0.
	⟨2:0⟩	Part revision for the odd ROM in a ROM pair.
16		ROM Size and Group Type: Bits ⟨7:4⟩ are the ROM size flag. It is a multiple of 64K bits (zero is illegal). For ROMs that don't have this byte as above, 64K bits is assumed. Bits ⟨3:0⟩ 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 ⟨21:14⟩ of the address of the ROM header byte 0, shifted right 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 through 8)
*                        byte checksum
*                 d3.w = 16-bit odd (bits 7 through 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
sumrom1   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     sumrom1      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 Series 200 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) 68xxx Family System File, SDF (Structured Disc Format) Boot Area, and UNIX[1] Boot Area[2]. All Boot ROM revisions support LIF. Only the 3.0 Boot ROM and later support all formats.

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).

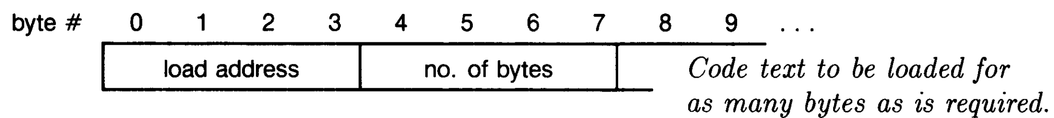
The disc formats are defined below by using Pascal structure type declarations. The following types are used in these definitions:

```
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 are the load address. The second 4 bytes are 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.)

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] The UNIX boot area is not necessarily the same format as that of HP-UX, Hewlett-Packard's implementation of UNIX.



Format For A Load Segment

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 Series 200 products as a common means for data communication.

File names must begin with "SYSTEM_" to be found by the Boot ROM. The 3.0 and later Boot ROMs also will find system files whose names begin with "SYS".

A LIF disc can have multiple system files on it. The 3.0 and later Boot ROMs allow any of them to be chosen (see Installation Guides for more detail). 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 (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;
```

The field names are defined as follows:

`LIFid`	equal to −32 768 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;
```

The field names are defined as follows:

`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 Series 200 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 (year/month/day/hour/minute/second) indicating the time of creation of the file. `LIFtoc` is ignored by the Boot ROM.
`LIFl_flag`	(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. First, a dump of sector 0:

```
     +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  ........
```

Hexadecimal/ASCII dump of sector 0 (LIF system record)

The field names are defined as follows:

`LIFid`	is −32 768 or `$8000`.
`LIFvolume_label`	is `'HP9826'` or `$485039383236`.
`LIFdir_start_address`	is sector 2 or `$00000002`.
`LIFoct_10000`	is 4096 or `$1000`.
`LIFdummy`	is 0 or `$0000`.
`LIFdir_length`	is 14 or `$0000000E`.
`LIFversion`	is 1 or `$0001`.
`LIFzero`	is 0 or `$0000`.

Following this information in sector 0, the next 3 4-byte quantities are defined as follows:

- Number of tracks per surface is 33 or `$00000021`.
- Number of surfaces per medium is 2 or `$00000002`.
- Number of sectors per track is 16 or `$00000010`.

Now, a dump of sector 1:

```
     +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  PL.B....
10   00 00 01 B1 00 00 00 00  ........
18   00 00 80 01 FF FE B7 A0  ........
20   63 62 61 63 6B 75 70 20  cbackup      Second entry
28   20 20 E8 14 00 00 01 C1    ......
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    ......
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    ......
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 ......
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
```

Hexadecimal/ASCII dump of sector 1 (LIF directory entries)

The first entry (bytes `$00` through `$19`) is for the 2.0 HPL soft system:

`LIFfile_name`	is `'SYSTEM_HPL'` or `$53595354454D5F48504C`.
`LIFfile_type`	is −5822 or `$E942`.
`LIFstart_address`	is 16 or `$00000010`.
`LIFfile_length`	is 433 or `$000001B1`.
`LIFtoc`	is 0 or `$000000000000`.
`LIFl_flag`	is 1 or most significant bit of `$8001`.
`LIFvol_number`	is 1 or least significant 15 bits of `$8001`.
`LIFimplement`	is `$FFFEB7A0`.

SDF Boot Area Format

Only the 3.0 and later Boot ROMs support Structured Disc Format (SDF).

SDF is a hierarchical directory structure that is used by the Shared Resource Manager (SRM) to manage its files. It is tree-like, hierarchical, and meets needs of larger file systems. Only the 3.0 and later Boot ROMs allow the SDF boot area to contain a 68xxx system. The purpose of this is to allow the SRM (which runs in a 68xxx 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 (again in Pascal) to be recognized by the Boot ROM:

```
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 and later Boot ROMs:

```
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;
```

The field names are defined as follows:

`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 Series 200 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. 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.

UNIX Boot Area Format

The UNIX 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.)

NOTE

The UNIX boot area is not necessarily the same format as that of HP-UX, Hewlett-Packard's implementation of UNIX.

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 or later 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;
```

The field names are defined as follows:

`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 Series 200 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.

All three of the formats (LIF, SDF, and UNIX) usually imply that they are present on some sort of sector-oriented mass storage device. The 3.0 and later Boot ROMs also allow 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 and later Boot ROMs can find soft systems stored in ROM/EPROM in the ROM address space (64K bytes through 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 16K byte 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 ROM address space. Unit 1 is the second ROM/EPROM pseudo-disc found in ROM address space. Unit *n* is the $(n+1)$th 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 reads from directly. The 3.0 and later Boot ROMs also have 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 resources. Among these sharable resources are soft operating systems.

Only the 3.0 and later Boot ROMs will search the `/SYSTEMS` directory of each SRM disc volume for Series 200 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 the *Filer* chapter of the *Pascal 3.0 Workstation System.*

This 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 unit specifier, is described.

Default Mass Storage

The Boot ROM is responsible for setting up a variable called the Default Mass Storage Unit Specifier, `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 address `$FFFFFEDC`.

`DEFAULT_MSUS` is used by systems for three purposes:

1. The default Mass Storage Unit Specifier by language systems (e.g., "`msi`" statement in HPL or "`MASS STORAGE IS`" 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 3.0).

`DEFAULT_MSUS` consists of four bytes:

- The *device type* byte, which specifies the mass storage device and media format.
- The *unit* byte, which specifies the drive number, unit number, or volume number.
- The *select code* byte, which specifies the I/O select code of the device.
- The *primary address* byte, which specifies the HP-IB address or node address.

Some of these bytes may not be 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 and later Boot ROMs were designed to provide for the future. Included in this category are: 7905, 7906, 7920, and 7925.

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.

TYPE Byte ($FFFFFEDC)

The TYPE byte has two fields. Bits ⟨7:5⟩ specify the directory format (e.g., LIF, SDF, Special). Bits ⟨4:0⟩ specify which device it is (e.g., internal 5.25″ floppy). A directory format of type Special is for boot devices that the Boot ROM communicates with in a special way (e.g., SRM, ROM).

TYPE⟨7:5⟩	– Directory Format
0	= LIF (Logical Interchange Format)
1	= SDF (Structured Directory 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	= Model 226/36 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	= 913X A 5-Mbyte 5.25″ winchester (HP-IB)
8	= 913X B 10-Mbyte 5.25″ winchester (HP-IB)
9	= 913X C 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 and Subset '80 devices (256-byte blocks; HP-IB)
17	= all other Command Set '80 and Subset '80 devices (HP-IB)
18–19	= Reserved by HP for Future
20	= ROM/HP 98255 EPROM pseudo-disc (ROM Space Memory)
21	= Reserved by HP for Future
22	= HP 98259 Bubble memory card
23–31	= Reserved by HP for Future

UNIT Byte ($FFFFFEDD)

The UNIT byte specifies the drive number on which a medium is found. Most devices such as a 9122 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 7906 can have two levels of specification: a volume number and a unit number. This happens, for example, if there are 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.

UNIT⟨7:0⟩ – Device Dependent Variant Record

```
UNIT_byte=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;
```

SELECT CODE Byte ($FFFFFEDE)

The SELECT CODE byte specifies the I/O card address of the boot device. This byte has a value between 0 and 31, inclusive. 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 and internal 5.25″ floppies do not have a select code).

PRIMARY ADDRESS Byte ($FFFFFEDF)

The PRIMARY ADDRESS byte is a device dependent address field. It has a different meaning depending upon the device. For some devices, ROM, for example, it is meaningless. 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 internal flexible disc drive 0.

The DEFAULT_MSUS is set according to the following algorithm:

1. The same mass storage medium or device from which the system was loaded, for all but ROM systems; or
2. If a ROM system is loaded, then, 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).
   b. First device found with media present in boot list (shown below) if ROM is specified in DEFAULT_MSUS. ("Media present" means that the media is installed, on-line, with the door closed, and is one of the following formats: LIF, SDF, or UNIX.)
   c. First device found present in the boot list (shown below) if no media can be found (an on-line disc drive with the door open fits this category).
   d. If no devices are present, then a LIF media in an 8290xM drive at HP-IB select code 7, bus address 0, drive 0. (This case can occur only on machines with no internal mass storage, when a ROM system is loaded, and no external mass storage devices are on-line.)

When searching for the `DEFAULT_MSUS`, the 3.0 and later Boot ROMs use 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:

1. Internal flexible disc drive 0
2. External discs at select codes 0-31, bus address 0, unit 0, volume 0
3. SRM at node 0 at select code 21 on volume 8
4. HP 98259 bubble memory card on select code 30
5. HP 98255 ROM/EPROM pseudo-disc, unit 0
6. ROM systems
7. Internal flexible disc drives 1 through *n*
8. Remaining external discs at select codes 0-31, bus addresses 0-7, units 0-16, volumes 0-7
9. Remaining SRMs at select codes 0-31
10. Remaining bubble memory cards on select codes 0 through 29 and 31
11. Remaining ROM/EPROM 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 checked before going to select code 8. The SRM equivalent of unit number, the volume, has the order of priority of units 8 through 24 in order, and there are no units 0 through 7.

CPU Board ID PROM

Several of Hewlett-Packard's Series 200 products contain a socketed software readable ID PROM (Programmable Read-Only Memory) with the serial number and product name in them.

To determine if the ID PROM is present, look at bit 0 (the least significant bit) of address `$FFFFFEDA` (also known as `SYSFLAG2`). A value of 1 means that a valid ID PROM of the format shown below is present. A value of 0 means that there is no ID PROM present, that the ID PROM did not checksum, or that the ID PROM is not Hewlett-Packard format.

Visually, one can tell that a machine has a ID PROM because the 3.0 and later Boot ROMs will display the serial number on the screen at powerup if an ID PROM is present.

The PROM is located at `$5F0001` when present and contains valid data in every other byte.

Address	Bytes Used	Item Description
`$5F0001`	002	Checksum
`$5F0005`	001	Size of ID PROM in multiples of 256 bytes (0 indicates 256 bytes)
`$5F0007`	011	Machine serial number represented in ASCII
`$5F001D`	007	Product number represented in ASCII
`$5F002B`	001	8 bits of processor board configuration; reserved at all 1s.
`$5F002D`	016	Required internal peripherals; allows up to 16 devices. Bits ⟨3:0⟩ of a byte indicate the type of peripheral required. Bits ⟨7:4⟩ give details on the required peripheral.

TYPE: ⟨3:0⟩	subfield: ⟨7:4⟩
0 = Reserved	
1 = Keyboard	0 = 98203B 1 = 98203A 2 = 46020A
2 = CRT alpha	which monitor (see `CRTID`
3 = HP-IB	
4 = Graphics	
5 = Mass Storage	0 = 1-drive minifloppy 1 = 2-drive minifloppy 2-15 = reserved
6 = DMA	
7 = Battery backup	
8 = CPU board	0 = MMU 1 = Timer 2 = Cache 3-15 = Reserved
9 = Bit-mapped display controller	0 = 9837A style
10-14 = Reserved	
15 = Nothing	

`$5F004D`	004	Required installed minimum amount of memory (bottom address)
`$5F0055`	032	Required I/O cards. Allows up to 16 devices; all `1`s indicates it is not used. `CARD`⟨12:8⟩ = ID of card; `CARD`⟨4:0⟩ = select code of the card.
`$5F0095`	004	Boot MSUS of device to try before Boot List scan.
`$5F009D`	017	Boot file name in ASCII to try before Boot List scan.
`$5F00BF`	004	Delay in milliseconds before Boot scan.
`$5F00C7`	001	Owner byte: zero means it is HP format
`$5F00C9`	001	ID PROM revision byte (currently 0)
	101 *(total used)*	Bytes allocated
`$5F00CB`	002	Spare checksum (set to `$FF`s) to allow second pass to ID PROM
`$5F00CF`	025	Reserved at `$FF`s for HP Future Use
`$5F0101`	128	Reserved at `$FF`'s for Future Use
	155 *(total left)*	Bytes reserved for future use (reserved at all `$FF`s):

Below is an example of a Model 226A with the serial number 2010A000000:

```
          +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  ........
```

Hexadecimal/ASCII dump of a Model 226A ID PROM

No two machines will have the same ID PROM (unless it's fraud or non-HP).

The ID PROM is checksummed to verify its validity (see section on ROM headers for checksum algorithm). A byte is used to determine the size of the ID PROM to allow for larger size ID PROMs in the future. The machine serial number is of the form DDDDCSSSSSS 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., 9826) is the number HP Marketing gives a machine. It is 7 characters long but may be blank padded. 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 in the power-up self-test process. The owner byte specifies who generated the ID PROM. The revision byte specifies what version the ID PROM format is. Much of the ID 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 ID PROM is just a small part of machine configuration identification done by operating systems that run on Hewlett-Packard's Series 200 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 Series 200 products.

Machine Configuration

Machine configuration identification is important on Hewlett-Packard's Series 200 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 Model 216, 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 a particular Model 216 has the CRT alpha highlights capability, the software could use it.)

SYSFLAG

`SYSFLAG` is a byte at address `$FFFFFED2` 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`⟨0:0⟩	Width of CRT alpha. 0 = CRT alpha is 80 characters wide. 1 = CRT alpha is 50 characters wide.
`SYSFLAG`⟨1:1⟩	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`⟨2:2⟩	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`⟨3:3⟩	Internal keyboard controller. 0 = internal keyboard controller present. 1 = internal keyboard controller not present.
`SYSFLAG`⟨4:4⟩	CRT configuration register (presented below). 0 = no CRT configuration register present. 1 = CRT configuration register present.
`SYSFLAG`⟨5:5⟩	Internal HP-IB. 0 = internal HP-IB present. 1 = internal HP-IB not present.
`SYSFLAG`⟨6:6⟩	0 = no working high-resolution CRT present (there may be a broken one, though). 1 = Working high-resolution CRT present.
`SYSFLAG`⟨7:7⟩	Reserved for HP, set to a value of 0.

SYSFLAG2

There is a second set of system configuration bits in the byte SYSFLAG2, at address $FFFFFEDA. SYSFLAG2 is also generated by the Boot ROM before a system is given control. SYSFLAG2 is defined as follows (bit 0 is the least significant bit):

SYSFLAG2⟨0:0⟩ ID 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.

SYSFLAG2⟨1:1⟩ Processor type. 0 = processor is not a 68000 (it's probably a 68010 or 68012). 1 = processor is a 68000.

SYSFLAG2⟨2:2⟩ Timer. 0 = processor board timer present. 1 = processor board timer not present.

SYSFLAG2⟨7:3⟩ Reserved for HP future use at value of 0100 1 binary.

Both SYSFLAG and SYSFLAG2 should be examined one 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

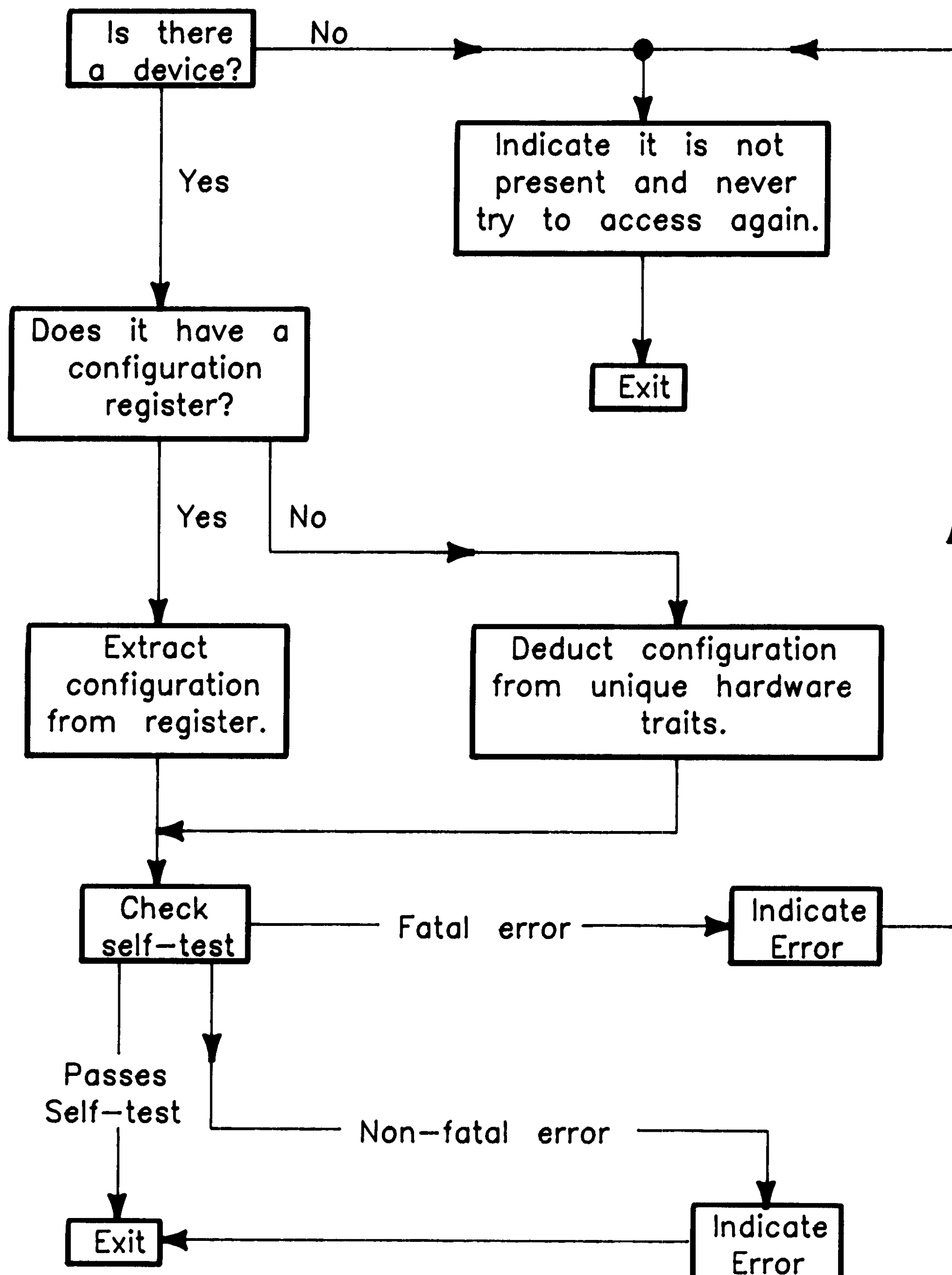
BATTERY is a byte at address $FFFFFDCD that contains a value of 1 if the battery backup hardware option is installed and a value of 0 otherwise. BATTERY is initialized by the Boot ROM at power-up. (The 3.0L Boot ROM will always set this byte to a value of 0 indicating no battery is present.)

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.

Device Configuration Identification

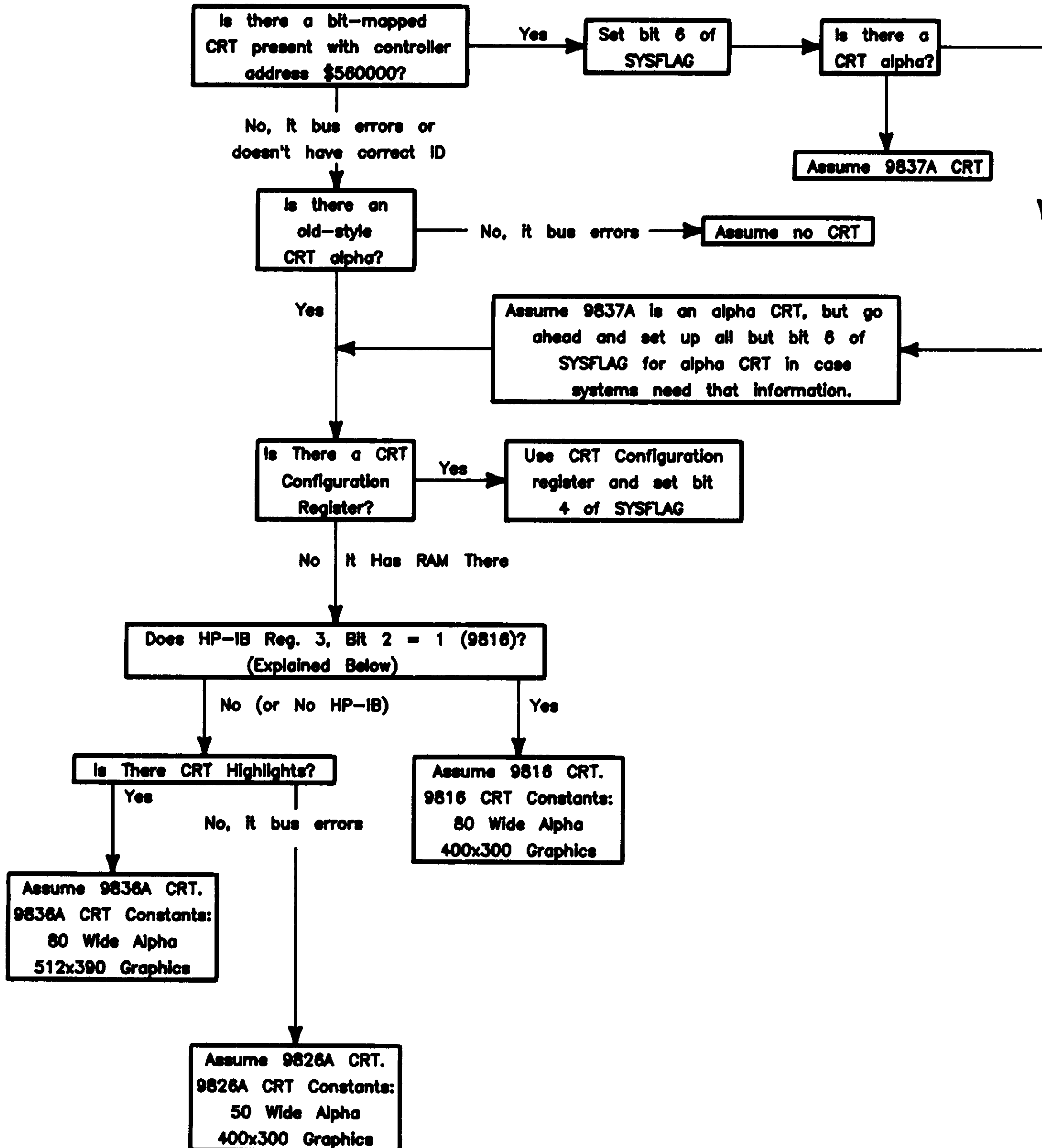
Below is the basic algorithm for determining a device's configuration and self-testing it. The actual device identification—for example, keyboard vs. CRT—is done by memory address. The initial test for the presence of a device is device-dependent by looking for any of four things:

- No bus errors at a particular address,
- Memory at a particular address,
- A ROM value at a particular address, and/or
- A certain read/write protocol at a particular address.



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 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 (Model 226 and Model 236). 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 Model 226, Model 236A, or Model 216.

First, a check should be made to see if the CRT is the same as a Model 216. This is done by looking to see if bit 2 of the byte at 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 Model 216-like CRT, then a check is made to see which of the two CRT tops it has (Model 226 or Model 236A). This is done by determining if it has alpha highlight capability. A Model 236A-like CRT has highlight capability and a Model 226-like CRT does not.

Below is the 16-bit CRT configuration identification register (at address `$51FFFE`; bit 0 is the least significant bit):

CRTID⟨15:15⟩	Self-Initializing CRT.
CRTID⟨14:14⟩	Reserved by HP for future, set to 0.
CRTID⟨13:13⟩	State of character mapping. 0 = The character ROM contains US-ASCII and Katakana and partial Roman extension fonts, where the latter two fonts are mapped together. This font requires mapping of the Roman extensions, and does not contain all Roman Extension characters (see definition of "old" character set on the first page of the *Displays* chapter). 1 = The character ROM requires no mapping. It could be Roman 8 or any 8-bit set. All characters are defined.
CRTID⟨12:11⟩	Graphics memory layout:
0	Monochrome.
1	4 memory planes starting at `$520000`; 4 bits/byte using 256K bytes; 1 byte/pixel (least significant 4 bits).
2	3 memory planes starting at `$528000`; 8 pixels/byte using 128K bytes; (unused 64K bytes at `$520000`).
3	8 memory planes starting at `$520000`; 8 bits/byte using 256K bytes; 1 byte/pixel.
CRTID⟨10:10⟩	Same as SYSFLAG⟨2:2⟩.
CRTID⟨9:9⟩	Same as SYSFLAG⟨1:1⟩.
CRTID⟨8:8⟩	Same as SYSFLAG⟨0:0⟩.
CRTID⟨7:4⟩	CRT Number: Implies CRT constants for Boot ROM initialization of CRT controller and physical dimensions of CRT.
0	Model 226A Monitor
1	Model 236A Monitor
2	Model 216 Monitor
3	Model 236C Monitor
4	Model 220 B/W Monitor
5	Model 220 Commercial Monitor
6–15	Reserved by HP
CRTID⟨3:3⟩	1 = 50 Hz. 0 = 60 Hz.
CRTID⟨2:0⟩	Model number (0 is for all systems based on the 6845 CRT controller chip.)

Any CRT top whose sub-type top bits do *not* indicate monochrome—(`CRTID`$\langle 12{:}11 \rangle \neq 0$)—may also have a control register (which is a 16-bit word at address `$51FFFC`) to control the CRT:

bit 0 (least significant)	0 = Turns Off Graphics
	1 = Turns On Graphics

Keyboard

The keyboard has additional configuration information, that is not documented elsewhere. 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 98203B keyboard already uses the value zero. The 98203A keyboard uses the value 1. (See the *Keyboards* chapter for more detail on the keyboards.)

NDRIVES

In addition to knowing which kind of keyboard is present, one may want to know how many internal flexible disc drives are present.

The byte `NDRIVES` at address `$FFFFFED8` 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., the Model 216). This location is not valid on 1.0 Boot ROMs. (The Boot ROM I.D. in the 16-bit word at 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 ID 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 address `$3FFC`. It is only valid on 3.0 or later revision Boot ROMs (for the 1.0 and 2.0 Boot ROMs, assume the value of `$0501`). To tell which Boot ROM one has, look at Boot ROM revision word, below. A layout of the configuration word is shown below (bit 0 is least significant):

Bits ⟨15:11⟩	Reserved by HP, set to zero
Bit ⟨10:10⟩	0 = Boot ROM has I/O vectors 125–255 defined. See section *Low ROM Map Exception Vectors*, later in this chapter, for more details.
Bit ⟨9:9⟩	0 = Boot ROM has character table at `$4000` defined. See section *Character Tables*, later in this chapter, for more details.
Bit ⟨8:8⟩	0 = Boot ROM has a read interface (see section appearing later on the *Read Interface and Secondary Loading*).
Bits ⟨7:5⟩	These bits are reserved by HP, set to zero.
Bits ⟨4:0⟩	These bits form a value that indicates the size of Boot ROM in 16K byte increments.

The Boot ROM revision word is a 16-bit word at address `$3FFE`. It is present in all revisions of the Boot ROM. Its possible values are:

Negative (−19492)	1.0 Boot ROM (Model 226 only).
1	2.0 Boot ROM (Model 226/Model 236 only).
3	3.0 and 3.0L Boot ROM.
4	4.0 Boot ROM.

To determine the difference between the 3.0 Boot ROM and the 3.0L Boot ROM, look at the Boot ROM size byte, which is located at `$3FFD`. This byte is equal to 3 for for 3.0 Boot ROM and it is equal to 1 for the 3.0L Boot ROM. 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 and later Boot ROMs have these). Currently the 1.0, 2.0, and 3.0L Boot ROMs are 16K bytes in size. The 3.0 Boot ROM is 48K bytes in size, and the 4.0 Boot ROM is approximately 50K bytes.

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.

Power-Up Options

Before turning on the machine, there are several switch configurations which can be used to modify power-up behaviour.

Memory Test Length

On any Series 200 machine except the Model 216, bit 1 of the CPU ID switch register controls the memory test length for the majority of main memory. A value of of 0 specifies the fast[3] memory test, and a value of 1 specifies the slow memory test. For the Model 16, the 3.0 and 4.0 Boot ROMs' default memory tests are different: the 3.0 Boot ROM defaults to the long memory test, and the 4.0 Boot ROM defaults to the short memory test.

CPU ID Reg⟨1⟩	Time for Test	Type of Memory Test
0	1 second per 256K bytes	Address and address complement
1	4 seconds per 256K bytes	Marching/walking 1s and 0s

Self-Test Looping

On all Series 200 machines, there is a switch that can direct the power-up self-test to repeat indefinitely. On the Model 16, this switch is bit 2 of the CPU ID register; on all other Series 200 machines, it is bit 0 of the CPU ID register. A value of 0 causes normal power-up; a value of one causes infinite self-test repetition.

When looping through self-test, the Boot ROM will not wait, beep, or stop for most failures. The following failures *will* stop self-test looping:

- A CPU register failure. In this case, the Boot ROM stops with the CRT blanked and the value `$81` (`1000 0001`) on the LEDs.
- A memory failure in the *top* 16K bytes. In this case, the Boot ROM stops with the message "`NEED GOOD RAM ABOVE FFC000`" on the CRT (unless it is a Model 237 CRT, in which case no message is displayed) and the value `$84` (`1000 0100`) on the LEDs.
- Boot ROM checksum failure. In this case, the Boot ROM stops with the message "`CONTINUE AT OWN RISK (Press RETURN to Continue)`" on the CRT[4], the value `$83` (`1000 0011`) on the LEDs, and then beeps four low tones and two high tones. Pressing `ENTER` (or `Return`) key causes it to continue self-test looping.

If there is a test stimulus board, it will be called at the beginning of each self-test loop, provided the test board returns to the Boot ROM through its return jump vector.

[3] All these memory-test times are based on a 68000 running at 8 MHz.
[4] Depending on the keyboard on your system, the message may say, "`CONTINUE AT OWN RISK (Press ENTER to Continue)`".

50/60 Hz CRT

The 50/60 Hz switch or jumper is found on the CRT alpha board for all Series 200 computers except the Models 216 and 237. A value of 0 selects 60 Hz, and a value of 1 selects 50 Hz.

On the Model 216, the 50/60 Hz switch is bit 8 of the switches closest to the hole in the back of the machine. Again, a value of 0 selects 60 Hz, and a value of 1 selects 50 Hz. This is used as the power-up default, and can be overridden by configure mode in all machines except the Models 226 and 237.

The 50/60 Hz switches on the Model 237 are the two low-order switches closest to the short edge of the CRT board. These switches choose which initialization region of the Model 237 ID/Initialization ROM would be used. Since the ROM may change, the meaning of these switches should be determined from the manual associated with your machine. For this reason, the software cannot override the 50/60 Hz selection on the Model 237.

Configure Mode Software Override

Configure Mode is a Boot ROM feature that allows the modification of some power-up defaults independent of which language will eventually be booted. To enter Configure Mode, press Ctrl-C before a system is booted.

All mass-storage operations lock out and defer the response to the Ctrl-C (just as they defer RESETs). This is to prevent leaving a mass storage device in an unusual state; for example, with the disc spinning.

Once in Configure Mode, options are displayed. If no key is depressed within approximately 5 minutes, the Boot ROM times out and starts the power-up sequence again. Regardless of the option chosen, most of the self-test will be repeated. If the memory test has been completed, and the 50/60 Hz CRT option is chosen, then the main memory test portion of the self-test will *not* be repeated.

Force Long Memory Test

If the T key is pressed, the self-test starts over and the long memory test is done. This takes 1 second per 64K bytes, and uses marching/walking 1s and 0s.

High-Resolution CRT RAM test

Pressing the R key causes the Boot ROM to execute a byte-wide RAM test on a high-resolution bit-mapped display CRT card at $560000 if one is present. The RAM test is visible on the CRT, and any errors are reported on both the CRT (if possible) and the LEDs. If an error occurs, then the Boot ROM stops until Reset is pressed. Pressing Reset restarts the booting process. This RAM test takes approximately 30 seconds for a 1M byte, single-plane frame buffer.

NOTE

This option in the Configure menu is present only if there is a bit-mapped CRT card at $560000.

50/60 Hz CRT

If the [5] or [6] key is pressed, a soft override of the power-up CRT frequency occurs.

NOTE

These options are not displayed on the Model 226, since it only supports 60 Hz.

NOTE

The Model 237 does not support software modification of the CRT frequency, so these options are not be shown even if there is an alpha CRT also present. If a [5] or a [6] is pressed during Configure Mode when there are both an alpha CRT and a high-resolution bit-mapped CRT, then the key pressed affects only the alpha CRT (even though the 5 and 6 options are not displayed). If only a high-resolution CRT is present, and a [5] or a [6] is pressed, then the Boot ROM re-enters self-test, but the CRT will not be affected.

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 address $FFFDC0).

Boot type	Value
POWER UP	0
CRASH RECOVERY	6
REQUESTED RE-BOOT	12
REQUESTED BOOT	18

2. Address of lowest usable RAM (determined by RAM test) is stored in LOWRAM at $FFFFFDCE.
3. The flag byte showing battery backup presence (BATTERY) is set up.
4. Interrupts will be disabled; that is, the status register will be set to $2700.
5. The hardware will be in a state of RESET except for the keyboard, battery backup, the internal flexible disc, and remote RS-232 card in the case of a remote console.
6. Keyboard interrupts are disabled.
7. The keyboard buffer may not be empty.
8. Battery backup (if present) is set to give 60 seconds of protection, which is the maximum possible.
9. All vectors 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 address $00880000 (by looking for the 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* for the handling of trap instructions.)
10. The stack pointer will be pointing to the top of the booter stack area (address $FFFFFDAC).
11. SYSFLAG and SYSFLAG2 will be set correctly.
12. The system CRT controller will be initialized to its power-up values. The CRT will be blanked out. For an alpha CRT, graphics memory will be zeroed and turned off. For a bit-mapped CRT, graphics memory will be zeroed.
13. The DEFAULT_MSUS at $FFFFFEDC will be set up.
14. NDRIVES will be set up.
15. 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 address $FFFFFDCE. Boot ROM 1.0 steals no space. Boot ROM 2.0 steals 32 bytes. The 3.0L Boot ROM steals 44 bytes. Boot ROM 3.0 and 4.0 steal 160 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 address `$FFFFFEDB`, 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 and 4.0 Boot ROMs have 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.

The read interface drivers are made available for secondary loader programs. Despite the common opinion which says that loaded systems should not depend on the Boot ROM for any support after they have gained control, there are three reasons why the Boot ROM's read drivers must be accessible:

- Some systems, such as UNIX, are only permitted to load a small (768 bytes) code segment at boot time. This is hardly enough space to expect it to be able to start up a system on its own. In this small amount of code, it cannot possibly have its own mass storage drivers.
- Some systems, such as Pascal, are designed to have the kernel, which has hardly any drivers at all, booted in first. It would like to have the Boot ROM read more of the operating system code in. In other words, some systems—by design— require a two-pass booting process.
- The third reason is, "If the code is there, why do I have to maintain my own copy?"

Where possible, it is suggested that you use "soft" drivers (as opposed to Boot ROM drivers) as then you are independent of future Boot ROM changes.

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 address $FFFFFEDC).
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.

The read interface is designed to be extendable to any mass storage or network. It allows simple random-sector reading of a disc, as well as reading sectors relative to an opened file.

On all 1.0 and 2.0 Boot ROMs, systems must use floppy drivers with DRV_KEY ($FFFFFEDB; byte-sized) set to FALSE to load more code. Systems can tell which Boot ROM is installed by looking at the Boot ROM ID.

All routines in the Boot ROM are callable from assembly language directly. Pascal 3.0 in many cases has entry points that handle environment changes and eventually call the routines. See the section *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,* 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 $14          Offset to required size
mb_ptr     equ $10          Offset to memory pointer
f_area     equ $FFFED4      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 1K bytes.

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(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 does the following actions:

1. It initializes the default mass storage device.
2. It initializes internal temporaries.
3. It determines if the device and media are present and returns true if they are.
4. It verifies that the device in the parameter MSUS is of the expected mass storage specifier format.
5. Resets an internal pointer used by `M_FOPEN` (explained below) is to the start of the directory.

This routine can escape with any of the possible errors (escape codes): 4 through 7.

This is usually the first routine to be called in a sequence of calls to the Read Interface.

CALLING FROM PASCAL 3.0 (See *Using Boot ROM Routines from Pascal*, close to the end of this chapter):

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_minit(msus:msustype):boolean; external;
```

CALLING FROM ASSEMBLY LANGUAGE:

```
default_msus equu $fffedc         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` through `n6`	are names that can be up to 16 characters each in length,
`<p1>` through `<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 3.0 (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;
```

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 3.0 (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 3.0 (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 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 address `$FFFFFED3`, 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 3.0 has a slightly different mechanism. Pascal 3.0 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 3.0 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, 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 3.0 (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 address `$FFFFFDD2` 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 3.0 (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 3.0 (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 3.0 (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 3.0 (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 3.0 (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 through 7, will not be displayed mnemonically.)

The format is:

```
{number MOD 1000},{number DIV 1000} text
```

For example, 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 address `$FFFFFDC2`) 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 address `$FFFFFDC0`) 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    $fffdc2           10-character system name
bootit  move.l #'SYST',sysname   Set first 4 chars of name "SYSTEM_P"
        move.l #'EM_P',4+sysname Set last 4 chars of name "SYSTEM_P"
        clr.b  8+sysname         Set the terminator
        trap   #11               Go into supervisor state
        jmp    $1C0              Do it
        end
```

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 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 3.0 is the only language which is not already in supervisor mode. (See trap instructions of MC68000 User's Manual, 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
*                                       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 pointer to allocated
*                                       memory
```

CALLING FROM PASCAL 3.0 (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 $fffedc         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 through 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 3.0 requires that the machine be put in supervisor mode first. (See MC68000 User's Manual, regarding trap instructions that do this.)

CALLING FROM PASCAL 3.0 (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. It is suggested that the user do the clear him/herself. Clearing an alpha CRT merely requires placing an ASCII space (chr(20)) in each character byte and an ASCII null (chr(0)) byte in each highlight byte.

Clearing a bit-mapped CRT requires merely placing 0s in all frame buffer locations. Note: depending on your application, you may or may not want to clear the undisplayed portion of the frame buffer, where the character fonts and possibly also the DGL fill patterns reside.

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*, 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 set up 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 Model 226.

The raster character patterns for the Model 226A 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 68xxx exception vectors in low ROM. 68xxx 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

Interrupt Levels

$FFFFFFBE	Interrupt Level 1
$FFFFFFB8	Interrupt Level 2
$FFFFFFB2	Interrupt Level 3
$FFFFFFAC	Interrupt Level 4
$FFFFFFA6	Interrupt Level 5
$FFFFFFA0	Interrupt Level 6
$FFFFFF9A	Interrupt Level 7

TRAP Instructions

`$FFFFFF94`	TRAP Instruction 0
`$FFFFFF8E`	TRAP Instruction 1
`$FFFFFF88`	TRAP Instruction 2
`$FFFFFF82`	TRAP Instruction 3
`$FFFFFF7C`	TRAP Instruction 4
`$FFFFFF76`	TRAP Instruction 5
`$FFFFFF70`	TRAP Instruction 6
`$FFFFFF6A`	TRAP Instruction 7
`$FFFFFF64`	TRAP Instruction 8
`$FFFFFF5E`	TRAP Instruction 9
`$FFFFFF58`	TRAP Instruction A
`$FFFFFF52`	TRAP Instruction B
`$FFFFFF4C`	TRAP Instruction C
`$FFFFFF46`	TRAP Instruction D
`$FFFFFF40`	TRAP Instruction E
`$FFFFFF3A`	TRAP Instruction F

The ROM has a facility for decoding NMI; if the language system routes NMIs through 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 Interrupt 0
$FFFFFF10	Vectored Interrupt 1
$FFFFFF0A	Vectored Interrupt 2
$FFFFFF04	Vectored Interrupt 3
$FFFFFEFE	Vectored Interrupt 4
$FFFFFEF8	Vectored Interrupt 5
$FFFFFEF2	Vectored Interrupt 6
$FFFFFEEC	Vectored Interrupt 7
$FFFFFEE6	Pseudo Vector (unassigned). This has no stated purpose.
$FFFFFEE0	Pseudo Vector (unassigned). This has no stated purpose.
$FFFFFEDC	DEFAULT_MSUS (Default Mass Storage Specifier). See the section *Default Mass Storage*, earlier in this chapter, for its definition.

Variables

$FFFFFEDB	DRVR_KEY	Disable reads to default MSUS through flexible disc (ignored by 3.0 and later Boot ROMs—use READ interface).
$FFFFFEDA	SYSFLAG2	This is a system configuration byte. See *Machine Configuration*, section *SYSFLAG2*, earlier in this chapter.
$FFFFFED9	RETRY	Used by Flexible Disc Drivers
$FFFFFED8	NDRIVES	This byte is the number of internal drives minus one (255 means no drives). Not valid on 1.0 Boot ROMs. See *Machine Configuration*, section *NDRIVES*, earlier in this chapter.
$FFFFFED4	F_AREA	This long-word points to an area in low RAM that is used for Boot ROM temporaries. The address of the first word above this area is contained in LOWRAM. Not valid on 1.0 Boot ROMs.

$FFFFFED3	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. See *Machine Configuration*, section *SYSFLAG*, earlier in this chapter.
$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.
$FFFFFDCC	BLFLAGS	This byte contains booter temporary flags:
⟨0:0⟩	SYSTEM NOT FOUND	1 = not found.
⟨1:1⟩	ERROR	1 = error during load.
⟨2:2⟩	SOFT SYSTEM ONLY	1 = load only soft system.
$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, if the file name is 5 characters, then character number 6 should be a zero (`chr(0)`). 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; 6 = crash recovery; 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		Other booter temporary variables. *Systems should be careful of using memory at this address or higher for temporaries.*
$FFFFFDAC		Top of Booter stack.
$FFFFFAC0		*Systems should never load over this location or higher.* The 3.0 and later Boot ROMs 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.

Address	Contents	
$0000	Power up RESET stack address	
$0004	Power up RESET start address (this goes to the Boot ROM)	
$0008	Bus Error routine; jumps to $FFFFFFFA unless special	
$000C	$FFFFFFF4	Address Error
$0010	$FFFFFFEE	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	$00000000	Unassigned Vectors (Motorola has reserved these)
$0060	$FFFFFF1C	Spurious Interrupt
$0064	$FFFFFFBE	Interrupt Level 1
$0068	$FFFFFFB8	Interrupt Level 2
$006C	$FFFFFFB2	Interrupt Level 3
$0070	$FFFFFFAC	Interrupt Level 4
$0074	$FFFFFFA6	Interrupt Level 5
$0078	$FFFFFFA0	Interrupt Level 6
$007C	$FFFFFF9A	Interrupt Level 7
$0080	$FFFFFF94	TRAP Instruction 0
$0084	$FFFFFF8E	TRAP Instruction 1
$0088	$FFFFFF88	TRAP Instruction 2
$008C	$FFFFFF82	TRAP Instruction 3
$0090	$FFFFFF7C	TRAP Instruction 4
$0094	$FFFFFF76	TRAP Instruction 5
$0098	$FFFFFF70	TRAP Instruction 6
$009C	$FFFFFF6A	TRAP Instruction 7
$00A0	$FFFFFF64	TRAP Instruction 8
$00A4	$FFFFFF5E	TRAP Instruction 9
$00A8	$FFFFFF58	TRAP Instruction 10

Address	Contents
$00AC	$FFFFFF52 TRAP Instruction 11
$00B0	$FFFFFF4C TRAP Instruction 12
$00B4	$FFFFFF46 TRAP Instruction 13
$00B8	$FFFFFF40 TRAP Instruction 14
$00BC	$FFFFFF3A TRAP Instruction 15
$00C0—00FF	$00000000
$0100	$FFFFFF16 Monitored Interrupt 0 (User Interrupt Vector)
$0104	$FFFFFF10 Monitored Interrupt 1 (User Interrupt Vector)
$0108	$FFFFFF0A Monitored Interrupt 2 (User Interrupt Vector)
$010C	$FFFFFF04 Monitored Interrupt 3 (User Interrupt Vector)
$0110	$FFFFFEFE Monitored Interrupt 4 (User Interrupt Vector)
$0114	$FFFFFEF8 Monitored Interrupt 5 (User Interrupt Vector)
$0118	$FFFFFEF2 Monitored Interrupt 6 (User Interrupt Vector)
$011C	$FFFFFEEC Monitored Interrupt 7 (User Interrupt Vector)
$0120	JMP Flexible Disc Sector Read and 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	JMP Flexible Disc read sector; no move to user buffer
$013C	JMP CRTINIT Initializes CRT registers R0—R9
$0140	JMP GO Start the system (initialize vectors, etc.)
$0144	JMP Flexible Disc Power On
$0148	JMP CRTCLEAR Initialize CRT; clear characters and graphics
$014C	JMP CRTBLANK Blank alpha RAM
$0150	JMP CRTMSG Send a string to a given line of the CRT
$0154	address of the keyboard translation table
$0158	JMP GRPHCLEAR Clear graphics RAM
$015C	JMP KBD_NO_INT Wait for keyboard interrupt to stop
$0160	JMP KBD_CWAIT Wait for last command to keyboard
$0164	JMP KBD_WCMD Write command to keyboard
$0168	JMP KBD_WDATA Write data to keyboard
$016C	JMP KBD_RDATA Read data from keyboard (polling)
$0170	JMP CHK_BATTERY Check for battery backup
$0174	JMP BAT_NO_INT Wait for battery interrupt to stop

Address	Contents
$0178	JMP BAT_CWAIT Wait for last command to battery
$017C	JMP BAT_WCMD Write a command to the battery
$0180	JMP BAT_WDATA Write data to the battery
$0184	JMP BAT_RDATA Read data from battery (polling)
$0188	JMP SOFTLOAD Find and load SYSNAME file
$018C	JMP LIFHEAD Verify LIF volume header
$0190	JMP FINDFILE Scan LIF directory arbitrary file name/type
$0194	JMP LOADFILE Load a file from a given sector
$0198	JMP BLOCKLOAD Load a block from a given sector
$019C	JMP END_TEST_ROM Entry from test ROM to continue boot
$01A0	JMP RTN_TO_MONITOR Entry for "RETURN" to RAM monitor (880xxx)
$01A4	JMP REQ_REBOOT Reload and start current system
$01A8	JMP BIN_TO_DEC Binary-to-decimal conversion
$01AC	JMP PNT8HEX Binary-to-hexadecimal conversion
$01B0	address NMI_DECODE Address of NMI decoder ISR
$01B4	JMP BOOT_BUSERROR Boot ROM bus error ISR
$01B8	JMP CRASH Crash recovery routine
$01BC	JMP FMSGS Flexible Disc error messages routine
$01C0	JMP REQ_BOOT Boot SYSNAME system
$01C4	JMP READ_KEYS Read from keyboard and translate (polling)
$01C8	JMP BEEP Cause the keyboard to beep
$01CC—$01EA	Floppy test utilities jump table

User Interrupt Vectors

New as of 3.0 Boot ROM. No hardware currently supports user interrupt vectors.

$01EC	$00808000	User interrupt vector 123. Reserved for Red board monitor.
$01F0	$00801D7C	User interrupt vector 124. Reserved for Red board monitor.
$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 3.0; and Boot ROM routines expect to run in Supervisor mode, while user programs in Pascal 3.0 run in User mode.

The Pascal operating system 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 3.0 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 (privileged instr. on 68010)
   RTS                    return to caller
   ```

3. If the Boot ROM routine accesses the internal minifloppy disc 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 3.0. 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 3.0 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 CLR I/O keys on the Pascal 3.0 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 I/O 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.

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 68xxx 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:
   a. Integer and floating point math
   b. String operations
   c. Set operations
   d. Heap management (to implement `NEW`, `DISPOSE`, `MARK`, `RELEASE`)
   e. File operations: `OPEN`, `CLOSE`, `READ`, `WRITE`, etc.
   f. Many, many more
10. Read the Boot ROM 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 3.0 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 resolvable 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 disc in `#3`.
5. Press `'B'` for boot command.
6. Enter `'#3:SYSTEM_X.'` for name of system file on minifloppy.
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

        moveq   #0,d1           clear highlight byte
again   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 68xxx 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 (may be used by HP in the future)
$FFFFFF46	13	−21	unassigned (may be used by HP in the future)
$FFFFFF4C	12	−21	unassigned (may be used by HP in the future)
$FFFFFF52	11	−21	supervisor mode
$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	IOresult <> 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

DGL Internals 12

Introduction

The History and Philosophy of DGL

The Device-independent Graphics Library (DGL) had its beginnings around 1976 at the University of Colorado. It was written in Fortran at the time. Hewlett-Packard perceived the package as having potential, and subsequently secured rights to the software. Having done this, DGL was ported to the HP1000 minicomputer, still in Fortran. Others soon saw the utility of the package, and more and more peripheral support was added—both input devices and output devices.

After the Series 200 computers came out, DGL was translated into Pascal. There were several design criteria that HP strove for in the conversion to the new language, and these were inspired, in part, by a desire to more closely adhere to the SIGGRAPH core standard. The design criteria were:

- Flexibility. The DGL package, as it is to be on the Series 200, should be flexible enough to be able to expand as new mainframes and graphics peripherals become available.
- Device independence. A nice, "friendly" graphics package should be able to send its output to this device or that device with very few and small modifications.

Hand in hand with the design criteria went the implementation criteria. That is, yes, we can translate it into Pascal, but how should we do it for *these* computers with *these* peripherals with *our* version of Pascal...? We came up with the following implementation criteria:

- Modularity. As DGL stands now, it is quite a modular package. That is, if you want to use a particular function, why should you have to load the whole package into your machine?
- Along with modularity comes the ability to add or remove device drivers in a relatively straightforward manner. If the DGL package is to be flexible, this is a must.
- In order to enact the latter of the design criteria above—device independence—a section of DGL had to be created to contain "tools." For example, how could the package be device-independent if some output devices had certain capabilities and other ones didn't? The "toolbox" simulates, in software, the lacking capabilities of the output device. Then during initialization, the required tools are automatically supplied by software.

These were accomplished, and the user-friendliness increased dramatically.

Currently, DGL exists on Series 200 Pascal Workstations (written in Pascal), Series 200 and 500 HP-UX (written in Fortran), and the HP1000 (Fortran).

Subsequent references to DGL in this chapter will refer to the Series 200 implementation, written in Pascal.

The Structure of DGL

DGL was written in several different levels of functionality. The highest level of the graphics software, called DGL, is the interface to the general user. It is best, if possible, to use this highest level of functionality to implement your programs and only use the lower levels when absolutely necessary. The topmost level of functionality is described in the *Pascal 3.0 Graphics Techniques* manual; please be familiar with that before tackling this chapter. This hierarchy makes it possible for people to use the package effectively whether they have very little programming experience, or are intimately familiar with the entire operating system.

For those with little familiarity with the system, there are high-level routines which do massive amounts of computation and/or graphics operations. These routines, invoked with a simple routine call, make assumptions and have default values for many of their parameters, enabling the user to get output without having to specify dozens upon dozens of parameters beforehand.

For those with substantially more knowledge of the system, all the parameters for which there are defaults can be explicitly controlled by the user's program. This allows very tight control of the operation of the system, and thus, the output.

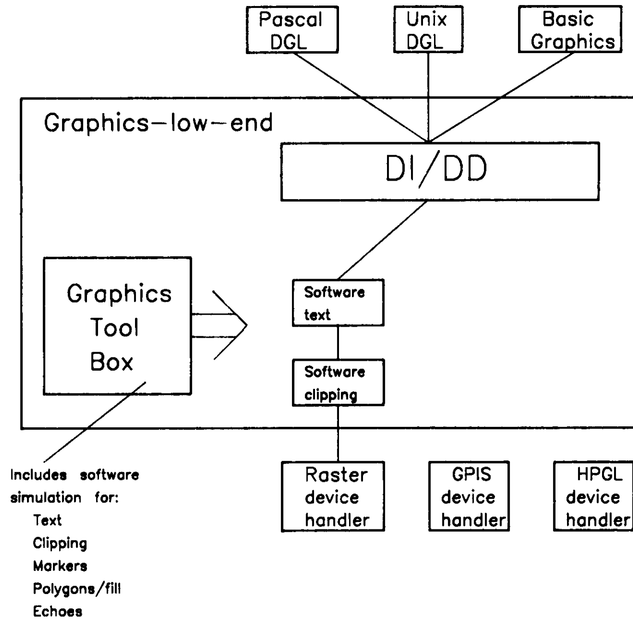
DGL This is the highest level of control. The routines at this level usually invoke sets of routines at the next lower level, GLE.

GLE Graphics Low-End (GLE) routines are routines which are much "closer to the metal" than the DGL routines, but also have more severe restrictions on what they can "understand." For example, DGL works in "user space," whereas GLE works in "display space;" DGL scales, GLE plots, etc. The DGL routines take care of the housekeeping functions needed to translate user-friendly entities into computer-friendly entities. For example, through DGL, the user can work in world coordinates, virtual coordinates, millimeters, or display units, whereas GLE can understand only display units.

Tools "Tools" are parts of the DGL system which accomplish necessary tasks, but may or may not be needed for a particular output device. For example, many HP plotters have on-board character generators, but most CRT displays do not. Thus, one tool would be the software character generator/stroke table; it is needed for some devices, but not for others.

Drivers Drivers are routines which actually do the bit- and byte-pushing required to command the various peripherals to do their thing. Every class of supported peripheral has its own driver, which is solely responsible for all communication between the computer and the peripheral.

This hierarchy is illustrated in the following diagram:



If a particular device supports "hardware" text generation, the "DI/DD"[1] box connects directly to the appropriate device handler (driver).

Since the driver calls are all implemented using the procedure-variable construct, you can create your own drivers from scratch, even to talk to a totally new peripheral.

[1] "DI/DD" stands for "Device-Independent/Device-Dependent." This is a layer of very small procedures which direct program flow to the driver for the currently-enabled peripheral.

IMPORT Hierarchy

Following is a structure diagram of the `IMPORT` hierarchy of the DGL library. The module `DGL_LIB` imports other modules, those modules import other modules, and those import still others. Below, indentation indicates which modules are imported by which other ones. Module names in boldface type are names which are referred to (by "*referenced above*") from farther down the list; the boldface type helps you find the names easily. Module names which are followed by a minus sign (−) do not import anything.

- **DGL_LIB**
  - DGL_TYPES –
  - SYSGLOBALS –
  - **SYSDEVS**
    - SYSGLOBALS –
  - **DGL_VARS**
    - DGL_TYPES –
    - GLE_TYPES –
  - DGL_AUTL
  - **DGL_GEN**
    - DGL_TYPES –
    - ASM
      - SYSGLOBALS –
    - DGL_VARS *(referenced above)*
    - **GLE_GEN**
      - GLE_TYPES –
    - GLE_TYPES –
  - GLE_TYPES –
  - GLE_GEN *(referenced above)*
  - **GLE_GENI**
    - GLE_TYPES –
  - DGL_CONFG_OUT
    - GLE_TYPES –
    - SYSDEVS *(referenced above)*
    - GLE_HPGL_OUT
      - GLE_TYPES –
      - **GLE_STEXT**
        - GLE_TYPES –
      - **GLE_ASTEXT**
        - GLE_TYPES –
      - **GLE_ASCLIP**
        - GLE_TYPES –
      - **GLE_SCLIP**
        - GLE_TYPES –
      - **GLE_SMARK**
        - GLE_TYPES –
        - GLE_AUTL –
      - **GLE_UTLS**
        - GLE_TYPES –
    - **GLE_RAS_OUT**
      - GLE_TYPES –
      - SYSGLOBALS –
      - SYSDEVS *(referenced above)*

    GLE_ARAS_OUT
        GLE_TYPES –
        SYSGLOBALS –
        SYSDEVS *(referenced above)*
        GLE_ASCLIP *(referenced above)*
        GLE_TYPES –
    GLE_STEXT *(referenced above)*
    GLE_ASTEXT *(referenced above)*
    GLE_SCLIP *(referenced above)*
    GLE_ASCLIP *(referenced above)*
    GLE_SMARK *(referenced above)*
    GLE_AUTL –
GLE_FILE_IO
    GLE_TYPES –
    GLE_UTLS *(referenced above)*
GLE_HPIB_IO
    GLE_TYPES –
    GENERAL_0
        IODECLARATIONS
            SYSGLOBALS –
        SYSGLOBALS –
        IOCOMASM
            IODECLARATIONS *(referenced above)*
    IOCOMASM *(referenced above)*
    IODECLARATIONS *(referenced above)*
    GLE_UTLS *(referenced above)*
GLE_UTLS *(referenced above)*
DGL_TOOLS
    GLE_TYPES –
SYSGLOBALS –
IODECLARATIONS *(referenced above)*
DGL_RASTER
    DGL_TYPES –
    DGL_VARS *(referenced above)*
    SYSDEVS *(referenced above)*
    ASM –
    SYSGLOBALS –
    GLE_TYPES –
    GLE_GEN *(referenced above)*
    GLE_AUTL –
    GLE_RAS_OUT *(referenced above)*
    DGL_GEN *(referenced above)*
DGL_HPGL
    DGL_TYPES –
    DGL_VARS *(referenced above)*
    GLE_GEN *(referenced above)*
    GLE_UTLS *(referenced above)*
    ASM *(referenced above)*
DGL_VARS *(referenced above)*

DGL_CONFG_IN
    GLE_TYPES –
    GLE_HPGL_IN
        GLE_TYPES –
        GLE_UTLS *(referenced above)*
    GLE_HPIB_IO *(referenced above)*
    GLE_KNOB_IN
        GLE_TYPES –
        GLE_UTLS *(referenced above)*
        SYSGLOBALS –
    GLE_UTLS *(referenced above)*
    DGL_TOOLS *(referenced above)*
    DGL_VARS *(referenced above)*
    SYSGLOBALS –
    DGL_KNOB
        DGL_TYPES –
        DGL_VARS *(referenced above)*
        GLE_TYPES –
        GLE_GEN *(referenced above)*
        GLE_GENI *(referenced above)*
        DGL_GEN *(referenced above)*
        DGL_CONFG_OUT *(referenced above)*
    DGL_HPGLI
        DGL_TYPES –
        DGL_VARS *(referenced above)*
        GLE_TYPES –
        GLE_GEN *(referenced above)*
        GLE_GENI *(referenced above)*
        DGL_GEN *(referenced above)*
DGL_TOOLS *(referenced above)*
DGL_IBODY
    DGL_VARS *(referenced above)*
IODECLARATIONS *(referenced above)*

DGL_INQ
    DGL_TYPES –
    DGL_VARS *(referenced above)*
    GLE_RAS_OUT *(referenced above)*
    DGL_GEN *(referenced above)*

DGL_POLY
    DGL_TYPES –
    DGL_VARS *(referenced above)*
    DGL_GEN *(referenced above)*
    GLE_TYPES –
    GLE_GEN *(referenced above)*
    DGL_LIB *(referenced above)*
    ASM –

DGL Modules

In general, if the first letter following the underscore in a DGL module name is an "A" (case is unimportant in module names), that module was written in assembly language. Otherwise, it was written in Pascal. This includes all levels of DGL: DGL, GLE, Tools, and Drivers.

DGL_TYPES	Contains declarations which must be imported by the graphics library.
DGL_VARS	Contains the graphics library global variables and types.
DGL_AUTL	Provides assembly-language utilities needed by DGL.
DGL_TOOLS	Provides general tools for DGL.
DGL_GEN	Contains most of the viewing transformations and color model transformations.
DGL_RASTER	Provides device-dependent initialization for raster devices, as well as raster utilities.
DGL_HPGL	Provides device-dependent initialization for Hewlett-Packard Graphics Language (HPGL) devices, input escape functions, and HPGL plotter setup utilities.
DGL_CONFG_OUT	Provides selection of different output device handlers.
DGL_KNOB	Provides device-dependent initialization for the knob.
DGL_HPGLI	Provides device-dependent initialization for Hewlett-Packard Graphics Language (HPGL) input devices.
DGL_CONFG_IN	Provides selection of different input device handlers.
DGL_LIB	Contains the normal user-interface level routines.
DGL_POLY	Contains the polygon routines.
DGL_INQ	Provides the user inquiry code for DGL.
DGL_IBODY	Sets global variables to a known (uninitialized) state.

Changes From Previous Implementations

The DGL package for Pascal 3.0 has been modified in several fundamental ways:

1. Pascal 2.x DGL used heap management extensively for dynamically allocating space for the initialization and termination of displays and locators. Allocating, deallocating and re-allocating space caused memory fragmentation to such an extent that memory overflows occurred in some programs (this only occurred when more than one display, or more than one locator were initialized during a single run). Pascal 3.0 DGL uses global variables which are allocated once at load time and remain until the program finishes. This satisfactorily reduces the chances for memory overflow errors.

2. Pascal 3.0 DGL is less dependent upon external libraries for support routines. Some of this was accomplished through duplicating certain procedures into the DGL routines so they are inherently available and an external library is unnecessary. The libraries which still need to be accessed are all in the standard `INITLIB`.

3. Several modifications have been made to drivers for communication with new peripherals.

DGL Responsibilities

As briefly mentioned in the introductory comments at the beginning of this chapter, the top layer of DGL is an easy-to-use interface between the user's program and the lower-level software. This is described in the *Pascal 3.0 Graphics Techniques* and *Pascal 3.0 Procedure Library* manuals. In the following section of this manual, we will further describe this interface.

DGL's Graphics Control Block

The main data structures of DGL are entities called the *graphics control blocks*, or GCBs. There are three of them: the `type` for one of them is defined in `DGL_VARS`, and is mainly used for the interface between the upper-level DGL routines and the low-level GLE routines. The `types` for the other two are defined in `GLE_TYPES`, and are for output and for input, respectively. They are described in the GLE section, later on in this chapter. Space for all the GCBs is allocated in `DGL_VARS`.

In this section, we will describe `GRAPHICS_CONTROL_BLOCK1`. Its purpose is to describe a display device, and it contains:

- Viewing transformation parameters. Several sets of parameters are used when DGL creates output. For example, `SET_ASPECT`, `SET_DISPLAY_LIM`, `SET_VIEWPORT`, and `SET_WINDOW` all affect `GCB^` fields and are involved in what the final picture looks like.
- Polygon attributes. When you draw a polygon, there are many factors involved: `SET_PGN_COLOR`, `SET_PGN_LS` (line style), `SET_PGN_STYLE`, `SET_PGN_TABLE` are all routines that affect these attributes. Others include `POLYGON`, `POLYGON_DEV_DEP`, `INT_POLYGON`, `INT_POLYGON_DD`.
- Data or instruction translation between DGL and GLE. For example, DGL defines its line styles differently than GLE, so there is an intermediate procedure (pointed to by a procedure variable in the GCB) which translates, or matches, the DGL definitions with the GLE definitions.

Concerning DGL's GCB:

- Its type is defined:

```
type
  graphics_control_block1=record
      . . .
  end;
```

- Its pointer is declared:

```
var
  gcb:        ^graphics_control_block1;
```

  Later, `GCB` is assigned the value `addr(GCB_SPACE)`.

- And its variable space is allocated:

```
var
  gcb_space:   graphics_control_block1;
```

All of the above are defined in, and exported from, module DGL_VARS.

A complete definition of GRAPHICS_CONTROL_BLOCK1 is in the *Pascal 3.0 Listings* manual, Volume II, under the heading *DGL_VARS.*

The DGL GCB is a global variable of DGL_VARS pointed to by GCB^.

Other DGL Variables

The following discussion addresses only a few of the DGL variables which are not contained in the graphics control block. As with the GCB, refer to the source listings for information not covered here.

DGL Initialization

When various stages of DGL are initialized, there are variables defined which contain information on the state of initialization of DGL and values indicating any errors which may have taken place. The following three are the first things which are initialized when GRAPHICS_INIT is executed. All three are boolean variables.

SYSTEM_INIT — Has the DGL system been initialized yet? This is set to TRUE by procedure GRAPHICS_INIT, assuming, of course, that it executed successfully. It is checked by every other user-level DGL routine.

DISP_INIT — Has a display, or output device, been specified? Every user-level output routine checks this. It is set to TRUE by procedure DISPLAY_INIT and DISPLAY_FINIT if execution was successful. If an error occurred, the error-return variable which comes back is set to the appropriate value.

LOC_INIT — Has a "locator" been defined? This is a graphics input function; therefore, every user-level graphics input routine checks this before continuing. It is set to TRUE by procedure LOCATOR_INIT if execution was successful. If an error occurred, the error-return variable which comes back is set to the appropriate value.

The reason that the phrase "user-level" was used above is that only the routines that the average user would access do the checking. It would be too time-consuming to do more, and it would be dangerous to do less. However, if you bypass the user-level routines and call lower-level routines, you must be sure the initialization was done properly, or else compromise the integrity of your system.

GRAPHICS_INIT also initializes numerous other variables; most of them in the GCB. Refer to the listings for specifics.

Other modules that are responsible for initialization operations are:

Module	Function
DGL_RASTER	DGL initialization for raster output devices.
DGL_HPGLI	DGL initialization for HPGL input devices.
DGL_HPGL	DGL initialization for HPGL output devices.
DGL_CONFG_OUT	Controls initialization of display devices.
DGL_CONFG_IN	Controls initialization of input devices.
DGL_IBODY	Initialization for the three booleans SYSTEM_INIT, DISP_INIT, and LOC_INIT. The initialization is done by the module body initialization entry point DGL_IBODY_DGL_IBODY.
DGL_LIB	Three routines in DGL_LIB do initialization: GRAPHICS_INIT, DISPLAY_INIT, and DISPLAY_FINIT. These were mentioned above.

DGL Errors

During initialization of the graphics system, if an error of some kind occurs, the user must be able to find out what error occurred. The following entities are involved in this.

GRAPHICS_ERROR_NUMBER
: This is a constant equal to −27 and is used with the ESCAPE procedure (see ERROR).

ERROR
: Routines detecting an error usually call the procedure ERROR, exported from DGL_GEN. This procedure sets GRAPHICS_ERROR (an integer, not a function) to the value passed, then does ESCAPE (GRAPHICS_ERROR_NUMBER).

GRAPHICS_ERROR
: This is an integer exported from DGL_VARS which contains the value of the most recent graphics error. This is an internal variable, not typically used by average users.

GRAPHICSERROR
: This integer function returns the value of the last graphics error. The various values and their interpretations are listed in the *Pascal 3.0 Workstation System* manual, under *Error Messages.* If ESCAPECODE= −27, the error was a graphics error. To determine the type of graphics error, the integer function GRAPHICSERROR (exported from DGL_LIB) returns the value of the integer GRAPHICS_ERROR. Note the presence or absence of the underscore in the two decidedly similar names. The function GRAPHICSERROR also resets the error number to 0.

GRAPHICSERROR Values

Below is a list of the graphics error numbers. These are constants, exported from DGL_VARS, and are the values returned by the integer function GRAPHICSERROR. Alongside the error numbers are the constant identifier names used internally by DGL. Note that these are constant identifiers, not elements in a enumerated type. The missing values are reserved for future definition.

0:		No error.
1:	ERR_SYS_INT	The system has not been initialized. You need to call GRAPHICS_INIT before calling any other graphics routines.
2:	ERR_DIS_INT	The display has not been initialized. You need to call DISPLAY_INIT before calling any output routines.
3:	ERR_LOC_INT	The locator has not been initialized. You need to call LOCATOR_INIT before calling any graphics input routines.
4:	ERR_ECHO_DIS_INT	You cannot use an echo before initializing a display.
6:	ERR_ASPECT	Illegal aspect ratio. One or more of the parameters was zero.
7:	ERR_BAD_PARMS	Illegal parameters. Catch-all error.
8:	ERR_OUT_PHYS	Parameters specified are outside physical display limits.
9:	ERR_OUT_WIND	Parameters specified are outside window limits.
10:	ERR_DISP_EQ_LOC	Loc limits given when the display and the locator are the same device.
11:	ERR_OUT_VIRT	Parameters specified are outside virtual coordinate limits.
12:	ERR_NO_DISPLAY_HARDWARE	Missing display hardware.
13:	ERR_OUT_LOC	Parameters specified are outside locator limits.
14:	ERR_NO_CTABLE	Current device does not support a color table.
18:	ERR_NEG_POINTS	The number of points in a polygon is less than zero.

Locator Echoes

When using a graphics input device, there is often a feedback loop consisting of the following parts:

- The user moves the graphics input device locator (a graphics tablet, for instance).
- The software translates the locator position into a graphics cursor position on the screen.
- The user, observing the screen, notices the effects of his hand's recent movements, determines if they are appropriate, and corrects if necessary.
- The correction, if any, is effected by moving the hand controlling the locator's stylus.

The second step in the above scenario is the one DGL is concerned with. There are four variables dealing with this. The two W- variables are reals and the two D- variables are integers. All four are exported from DGL_VARS, and used, among other places, in DGL_LIB.

For the locator device:	D_LOC_ECHO_X D_LOC_ECHO_Y
For the feedback device:	W_LOC_ECHO_X W_LOC_ECHO_Y

There are two sets because for some operations, both a device-coordinate position and a world-coordinate position are needed. For example, both sets are needed when a snap-to-grid echo is used[2]. This is because the device-dependent values are needed as globals during the echoing, but world-coordinate values are needed as the values to return.

The translation takes place in the procedure SET_ECHO_POS, exported from DGL_LIB.

See DGL_LIB for specifics on the interface between DGL and GLE.

Specific DGL Tasks

Viewing Transformations

There are four user-level routines which are involved in defining the viewing transformation.

SET_DISPLAY_LIM
: This sets the logical display limits of the physical display device. After calculating and validating the new display limits, this calls DISPLAY_LIMITS (exported from DGL_GEN). This, in turn, defines the X and Y limits of GCB^.LOG_DISP_LIM, the ranges in both X and Y, and aspect ratio of the logical display device. Finally, it calls CALCULATE_VIEWING, which calculates a new viewing transformation, and flags the fact that the text drawing transformation needs to be recalculated before drawing text.

SET_ASPECT
: This sets the aspect ratio (Y size:X size ratio) of the subset of the screen selected by SET_DISPLAY_LIM. After validating the parameters, the aspect ratio is calculated, GCB^.CUR_VIR_LIM.XLIM and GCB^.CUR_VIR_LIM.YLIM are set to the normalized aspect ratio, and SET_VIEWPORT is called with parameters of 0.0 to the appropriate maxima in GCB^.CUR_VIR_LIM.

SET_VIEWPORT
: This sets the "viewport," which must be within the virtual coordinate system defined by SET_ASPECT. After validating the input parameters, SET_VIEWPORT sets the bounds in GCB^.VIEWPORT_LIM. Finally, it calls CALCULATE_VIEWING, which calculates a new viewing transformation, and sets CALC_TEXT_XFORM to TRUE, which tells the procedure GTEXT to recalculate the character drawing transformation before drawing any characters.

SET_WINDOW
: This sets the "window," which is the user-defined, or "world," coordinate system. After validating the input parameters, SET_WINDOW sets the bounds in GCB^.WINDOW_LIM. Finally, it calls CALCULATE_VIEWING, which calculates a new viewing transformation, and sets CALC_TEXT_XFORM to TRUE, which tells the procedure GTEXT to recalculate the character drawing transformation.

For further information, see the routines themselves in DGL_LIB.

[2] A snap-to-grid causes the visual feedback to be rounded to the nearest unit; thus, the graphics cursor has a jerky appearance as the graphics input stylus describes a smooth motion.

Color Space Conversion

The DGL library understands color specification in two different methods:

RGB The RGB system is a cubic color space in which the three degrees of freedom are the **R**ed, **G**reen, and **B**lue axes, all perpendicular to each other, and all ranging in value from zero to one. Gray shades are specified by having all three values equal.

HSL The HSL system is a cylindrical color space in which the three degrees of freedom are **H**ue, **S**aturation, and **L**uminosity. They are described as follows:

> Hue is the shade or tint of the color. This parameter ranges from zero to one, and is the direction out from the center of the cylinder. Being circular, a hue of zero is the same as a hue of one. Hues, from zero to one by sixths are: red, yellow, green, cyan, blue, magenta, and back to red.
>
> Saturation is the amount of hue which is added to white, and it also ranges from zero to one. A saturation of zero means there is no hue added to white; thus, when saturation is zero, hue does not have any effect. A saturation of one results in a pure, intense color.
>
> Luminosity—or "lightness," as it is sometimes called—specifies how much light is emitted by each pixel. Ranging from zero to one, a luminosity of zero results in black, regardless of hue or saturation. A luminosity of one results in the brightest color (note that in this context, a "bright" color is not the same as a saturated color).

The DGL module `DGL_GEN` exports two routines, called `CONVERT_RGB_TO_HSL` and `CONVERT_HSL_TO_RGB` which effect the conversion.

GTEXT Conversion

When writing text to a graphics output device through the procedure `GTEXT`, the GCB fields `DGL_CHAR_WIDTH` and `DGL_CHAR_HEIGHT` are involved in specifying the character size. Two more GCB fields, `CHAR_ROT_H` and `CHAR_ROT_W` control the character rotation. If `CALC_TEXT_XFORM` is `TRUE`, the character-drawing transformation is set up using these values and `CALC_TEXT_XFORM` is set to `FALSE` before any characters are drawn.

`GTEXT`, a DGL routine calls the GLE routine `GLE_TEXT`, which actually does the writing. `GLE_GCB^.info_ptr1` is set to the address of the first (*not* the zeroeth) character in the string. Note that this is the address of the first *printed* character, and not the ghost byte at the beginning of the string which specifies the current length of the string. `GLE_GCB^.INFO1` gets the length of the string to print. After these two fields are defined, `GLE_TEXT` is called with the single parameter `GLE_GCB`.

GLE Responsibilities

GLE, standing for Graphics Low-End, is the set of low-level routines underneath DGL. As mentioned before, DGL takes care of translating information into the forms that GLE can understand. One of GLE's advantages is its generality with respect to device type; that is, it presents a "standard" interface to all graphics peripherals (albeit without scaling transformations).

What Is GLE?

The Graphics Low End is interfaced through a device-independent layer of procedures which are very small and specialized (this layer of procedures was indicated by the DI/DD box in the diagram earlier). These procedures' purpose is directing program flow to the correct procedure for the currently enabled device. Depending on the device's capabilities, the correct program flow could be to a GLE tool[3] module, or perhaps to a device handler module.

Suppose you have configured your DGL system to work with a raster display. When the display is initialized (by `DISPLAY_INIT`), state information is saved indicating which functions the display module can perform, and which functions must be emulated by the GLE tools. For raster displays, text and clipping are not supported by the raster display module. When the GLE text routine is called, the DI/DD text procedure calls on the software text procedure from the GLE tool box to produce stroked, or vector, text. The software text procedure calls the software clipping routine which, in turn, passes clipped vectors to the raster device handler.

In the case of a HPGL device, the device can probably generate text in hardware, and can perform clipping in hardware. Thus, an HPGL driver need not do clipping or text generation in software; it can just send the commands which tell the output device to do the operation. However, since DGL strives to create graphics images which are as similar as possible on all display devices, it uses software tools to ensure that the output will be very consistent.

The tools provided from the Graphics Low End may be called from within GLE or may be called from the user level. An example of this would be to call the software text procedure even if the device supports hardware text. An advantage of this use would be to provide exactly the same text font on either a CRT or a plotter.

Device Independence

GLE provides a device-independent interface to a set of graphics devices. "Device-independent" is defined in this context to mean that the same results (or a reasonable approximation) can be achieved by all devices that support a given feature. It *does not* mean that passing the same parameters to a given function will produce the same results.

For example, all coordinates passed to GLE are in device-dependent units. However, a inquiry exists to determine the maximum device units for the enabled device, allowing the higher-level language to perform scaling in a device-*in*dependent manner. For display devices, the resulting picture will be essentially the same no matter which display device is used, even though the coordinates can be very different for different devices.

[3] As mentioned before, GLE tools include software simulation for text, clipping, markers, and echoes.

GLE's Graphics Control Blocks

It was mentioned in the DGL section that the main data structures of DGL are entities called the *graphics control blocks*, or GCBs, and that there are three of them: one for DGL and two for GLE[4]. In this section, we will talk about the GLE GCBs. The types are called GRAPHICS_CONTROL_BLOCK and GRAPHICS_INPUT_CONTROL_BLOCK, are defined in GLE_TYPES, and are for output and for input, respectively.

First, we will describe GRAPHICS_CONTROL_BLOCK. Its purpose is to describe a display device and maintain several variables which describe current output characteristics. It contains:

- Procedure variables. These are pointers to the drivers.
- Variables. These are used by the operations, and can have hardware-dependent or dynamically-changing data.
- Pointers. These point to system information; for example, I/O-related data.

The Output GCB

Concerning GLE's output GCB:

- Its type is defined in GLE_TYPES:

```
type
  graphics_control_block=packed record
      . . .
  end;
```

- Its pointer type is declared in GLE_TYPES:

```
type
  graphics_control_block_ptr:       ^graphics_control_block;
```

  Its pointer variable is allocated in DGL_VARS:

```
var
  gle_gcb:     graphics_control_block_ptr;
```

- Its variable space is allocated in DGL_VARS:

```
var
  gle_gcb_space:   graphics_control_block;
```

  Later, GLE_GCB is assigned the value addr(GLE_GCB_SPACE).

The output GCB is defined in, and exported from, module GLE_TYPES. See that section of the *Pascal 3.0 Listings*, Volume II.

[4] Actually there are more than three, but the other ones are very small and extremely specialized. We'll cover them later.

The following program prints the sine and cosine tables calculated by GLE_ASTEXT. After the character size and rotation are specified, "HELLO" is plotted, because the sine/cosine tables are not calculated until something is actually plotted. This way, unnecessary recalculation is minimized.

```
PROGRAM TRANSFORM(INPUT,OUTPUT);

$SEARCH '*GRAPHICS.'$

IMPORT DGL_LIB, GLE_TYPES, DGL_VARS;

VAR ERR   : INTEGER;
    I     : INTEGER;
    XRUN,
    YRISE : REAL;

BEGIN
 GRAPHICS_INIT;
 DISPLAY_INIT(3,0,ERR);
 SET_CHAR_SIZE(0.1,0.1);
 WRITE('X-RUN, Y-RISE ? ');
 READLN(XRUN,YRISE);
 SET_TEXT_ROT(XRUN,YRISE);
 WRITELN(#12);
 WRITELN;
 MOVE(-0.5,-0.5);
 GTEXT('HELLO');
 WITH GLE_GCB^ DO
  BEGIN
   FOR I:=0 TO 7 DO
    WRITE(COSX_TABLE[I]:4,' :  ');
   WRITELN;
   WRITELN;
   FOR I:=0 TO 7 DO
    WRITE(SINX_TABLE[I]:4,' :  ');
   WRITELN;
   WRITELN;
   FOR I:=0 TO 15 DO
    WRITE(COSY_TABLE[I]:4,' :  ');
   WRITELN;
   WRITELN;
   FOR I:=0 TO 15 DO
    WRITE(SINY_TABLE[I]:4,' :  ');
   WRITELN;
   WRITELN;
  END;
END.
```

Computation of these tables is done by the routine GLE_TEXT_XFORM in module GLE_STEXT. Whenever the user specifies a character size or direction, the CALC_TEXT_XFORM field of the GLE_GCB is set to 1 to indicate that the tables need recalculation. After the tables have been recomputed by GLE_TEXT_XFORM, CALC_TEXT_XFORM is reset to 0. Recalculation is done only when necessary; when the routine GLE_SOFT_TEXT in module GLE_ASTEXT finds CALC_TEXT_XFORM=1, it calls GLE_TEXT_XFORM via GLE_GCB^.CALC_SOFT_TEXT_XFORM.

The Input GCB

Concerning GLE's input GCB:

- Its type is defined in GLE_TYPES:

```
type
  graphics_input_control_block=packed record
      . . .
  end;
```

- Its pointer type is declared in GLE_TYPES:

```
type
  graphics_input_control_block_ptr:   ^graphics_input_control_block;
```

  Its pointer variable is declared in DGL_VARS:

```
var
  gle_gcbi:     graphics_input_control_block_ptr;
```

- Its variable space is allocated:

```
var
  gle_gcbi_space:   graphics_input_control_block;
```

  Later, GLE_GCBI is assigned the value addr(GLE_GCBI_SPACE).

The input GCB type is also exported from GLE_TYPES. See that section in the *Pascal 3.0 Listings*, Volume II.

Note that in both the graphics control blocks for GLE, boolean variables are not used. Instead, type GLE_SHORTINT is used with the values 0 meaning false, and 1 meaning true.

GLE Modules

Here is a list of the modules comprising GLE, along with a short description.

GLE_AUTL	Low-level bit manipulation for GLE routines.
GLE_UTLS	General tools (read, write, match) for GLE routines.
GLE_TYPES	Defines the Graphics Control Block (GCB) used by the GLE routines.
GLE_STROKE	Provides the stroke tables for the GLE software character sets. Used by GLE_ASTEXT.
GLE_ASTEXT	Assembly language software text generator (GLE tool).
GLE_STEXT	GLE software text setup routines (GLE tool).
GLE_SMARK	GLE software marker routines.
GLE_SCLIP	GLE software clipping routines.
GLE_FILE_IO	General file drivers for ASCII device handlers.
GLE_HPIB_IO	I/O routines for ASCII device handlers (part of driver).
GLE_HPGL_OUT	Device-dependent routines to drive HPGL output (driver).
GLE_HPGL_IN	Device-dependent routines to drive HPGL input (driver).
GLE_RAS_OUT	Device handler routines for raster devices (part of driver).

GLE_KNOB_IN	Provides a device handler for the knob input device (driver).
GLE_GEN	The interface between the GLE caller and the GCB output procedure variables. Also provides GLE_INIT_GCB to set output GCBs to an "uninitialized" state.
GLE_GENI	The interface between the GLE caller and the GCB input procedure variables. Also provides GLE_INIT_INPUT_GCB to set input GCBs to an "uninitialized" state.
GLE_ASCLIP	Assembly language clipping routines (GLE tool).
GLE_ARAS_OUT	Assembly language raster device driver[5].

GLE Initialization

The GLE modules include routines in both Pascal and assembly language. Following are the routines which include some sort of initialization functions.

GLE_KNOB_IN

This module exports a routine called GLE_INIT_KNOB_INPUT which deals with using the Rotary Pulse Generator (RPG or "knob") as a graphics input device. The knob, as a graphics input device, can be rotated clockwise or counterclockwise, either unshifted, affording ±X motion, or shifted, affording ±Y motion. The routine GLE_INIT_KNOB_INPUT does, among other things, the following:

- Defines INPUT_HANDLER_NAME (a field of the GCBI) to 'KNOB'.
- Associates several procedure-variable drivers to knob-related routines, also defined in the module.
- Defines the INPUT_NAME field of the GCBI to the mainframe being used, along with the input resolution of the device.

Also exported are two data types: a record which describes the knob state, and a pointer to it.

GLE_HPGL_IN

This module exports a procedure named GLE_INIT_HPGL_INPUT, along with some constants and type declarations. HPGL stands for Hewlett-Packard Graphics Language, and it is a language composed of two-letter mnemonics, parameters, and semicolon or carriage-return delimiters. HPGL input can take place from hard-copy plotters, but is more likely to be done from a graphics tablet or digitizer. The procedure GLE_INIT_HPGL_INPUT in module GLE_HPGL_IN does the following things, among others:

- Redefines the ISC_TABLE[SELECT_CODE].USER_TIME timeout value to 500 milliseconds.
- Attempts to contact the HPGL device. If successful, the routine continues; if not, set GCBI^.ERROR_RETURN to 1 and return.

[5] This listing is suppressed from the Assembly language listings volume due to the proprietary nature of software.

- Requests the HPGL device model number. If the number returned is unknown (for example, for a newly-introduced device), the device is treated as a generic HPGL plotter, ostensibly a 9872A—now obsolete. Note that the 9872A hardware is unsupported and does not work with Pascal 3.0 DGL. The generic specifier "9872A" usually does work but this does not imply formal support for the device.
- Associates several procedure-variable drivers to appropriate routines, also defined in the module.
- Restores the value of `ISC_TABLE[SELECT_CODE].USER_TIME` to its original value.

GLE_HPGL_OUT

This module exports a procedure named `gle_init_hpgl_output`, along with some constants and type declarations. HPGL output deals with hard-copy plotters. The procedure `gle_init_hpgl_output` does the following things, among others:

- Associates several procedure-variable drivers to HPGL-related routines, also defined in the module.
- Redefines the `ISC_TABLE[SELECT_CODE].USER_TIME` timeout value to 500 milliseconds.
- Attempts to contact the HPGL device. If successful, the routine continues; if not, set `GCBi^.error_return` to 1 and return.
- Requests the HPGL device model number. If the number returned is unknown (for example, for a newly-introduced device), the device is treated as a generic HPGL 4-pen plotter, ostensibly a 9872A—now obsolete. Note that the 9872A hardware is unsupported and does not work with Pascal 3.0 DGL. The generic specifier "9872A" usually does work but this does not imply formal support for the device.
- Restores the value of `ISC_TABLE[SELECT_CODE].USER_TIME` to its original value.
- Sets the plotter to the default condition, selecting pen 1: `DF;SP1;IM30;`.
- Finally, depending on the kind of device, various other parameters are defined. These parameters include:
  `PALLETTE`: How many colors are available to work with?
  `CONT_LINESTYLES`: How many "continuous" line styles are supported?
  `VECT_LINESTYLES`: Home many "vector-adjusted" line styles are supported?

GLE_ARAS_OUT

`GLE_ARAS_OUT`[6] exports an assembly routine called `RGCBINIT`. `GLE_ARAS_OUT` is the "guts" of the raster driver. It handles all supported displays. `RGCBINIT` initializes a GLE GCB for operation with a raster device, and among other things, does the following:

- It says that the raster device supports a complementing pen, non-dominant drawing, line erasure, dithering, polygons (including solid-filling), and drawing in the background color.
- Initializes to zero: vector linestyles, current pen and cursor position, current cursor state, etc.

[6] This listing is suppressed from the Assembly language listings volume due to the proprietary nature of software.

- Determines what kind of mainframe is being used, and initializes mainframe-dependent information.
- It initializes the RASTER_DEVICE_REC[7], which describes information unique to the raster device.

GLE_RAS_OUT

This is a Pascal module which exports many constants and types dealing with raster devices. The largest type, a record, could be called a GCB for raster devices. Its definition is given below:

```
type
  raster_byte=          0..255;
  raster_code_space=    packed array[1..240] of raster_byte;
  dither_type=          packed array [0..15] of raster_byte;
  cmap_def=             packed record
                          map_red:  gle_shortint;
                          map_grn:  gle_shortint;
                          map_blu:  gle_shortint;
                        end;
  system_cmap_def=      packed array [0..15] of cmap_def;
  raster_device_rec_ptr=^raster_device_rec;
  raster_device_rec=    record
    addr1:              anyptr;
    addr2:              anyptr;
    addr3:              anyptr;
    n3:                 gle_shortint;
    devicetype:         gle_shortint;
    deviceaddress:      integer;
    monitortype:        gle_shortint;
    plane1_addr:        anyptr;
    plane1_offset:      integer;
    plane2_offset:      integer;
    plane3_offset:      integer;
    n_glines:           gle_shortint;
    gspacing:           gle_shortint;
    bytesperline:       gle_shortint;
    hard_xmax:          gle_shortint;
    hard_ymax:          gle_shortint;
    red_intensity:      gle_shortint;
    grn_intensity:      gle_shortint;
    blu_intensity:      gle_shortint;
    dither_pattern:     dither_type;
    cursor_x:           gle_shortint;
    cursor_y:           gle_shortint;
    area_draw_mode:     gle_shortint;
    pen_draw_mode:      gle_shortint;
    linepattern:        gle_shortint;
    pen_number:         gle_shortint;
    cpen:               gle_shortint;
    oldpattern:         gle_shortint;
    rgltemp1:           integer;
    rgltemp2:           integer;
    rgltemp3:           integer;
    rgltemp4:           integer;
```

[7] The RASTER_DEVICE_REC is sometimes referred to in assembly language programs as RGL_GCB—raster graphics level GCB.

```
    rgltemp5:           integer;
    repeatrate:         gle_shortint;
    repeatcount:        gle_shortint;
    index:              integer;
    softvec:            raster_code_space;
    system_cmap:        system_cmap_def;
    brightness_sequence:packed array [0..15] of gle_shortint;
    count:              packed array [0..15] of gle_shortint;
    cmap_address:       integer;
  end;
```

Should you need to implement a raster driver or other driver for DGL, examination of the above data structure may be instructive.

Also exported from this module are two routines:

- GLE_INIT_RASTER_OUTPUT This routine does software and hardware initialization for all raster devices.
- GATOR_CLEAR This does a clear-screen operation for the high-resolution Model 237 display.

Included in the things that the procedure GLE_INIT_RASTER_OUTPUT does are:

- Calls RGCB_INIT (exported from GLE_ARAS_OUT).
- Associates procedure-variable driver names with routines.
- Depending on the hardware, other variables like the display name and the X and Y resolution are defined.
- Sets drawing mode and area-fill drawing mode to zero.

GLE_HPIB

This module provides a small subset of the functionality of the routines contained in the I/O library, so that graphics functions will not need to have the library IO on-line during compiles. The procedures that are exported are:

```
hpib_init
hpib_inq_timeout
hpib_set_timeout
hpib_write
hpib_read
hpib_term
```

Two other important (though not exported) routines are:

```
addr_to_talk
addr_to_listen
```

These routines try to address an HP-IB if the parameter 'DEVICE' looks like an HP-IB "device specifier" (e.g., 705). If 'DEVICE' looks like a select code, these functions are no-ops, except if the card at the select code is an HP-IB, in which case the I/O card is checked to see if it is *not* active controller and wait until the card is addressed appropriately by the controller.

These routines ultimately work by calling the I/O drivers pointed to by the ISC_TABLE, and therefore, the appropriate driver (e.g., HP-IB) must be installed for GLE_HPIB to work.

Features of GLE

I/O Independence

DGL (specifically, module GLE_HPIB) supplies standard I/O routines to GLE. These are small interface procedures which perform the I/O by calling system I/O drivers for the select code through the ISC_TABLE and by using GLE_HPIB utilities. The procedures are used by GLE via procedure variables in the appropriate GCB.

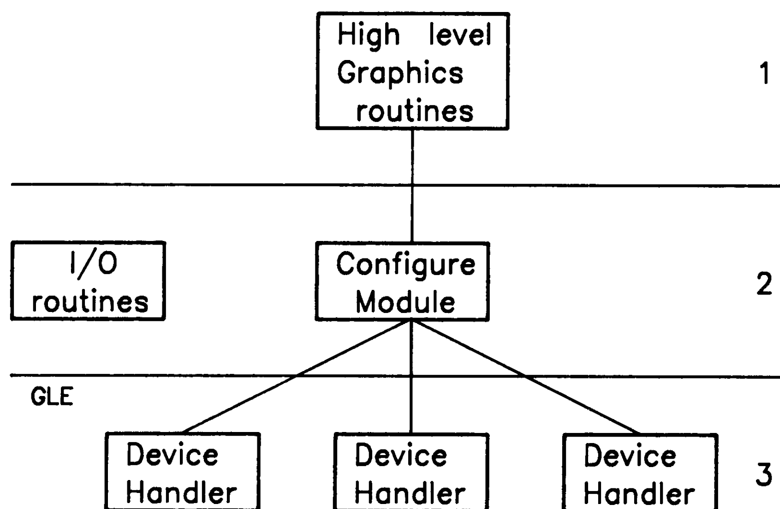
Not all device handlers use these I/O procedures. For example, HPGL devices use them to talk to HP-IB, but the raster device handlers do not need them.

This method of handling I/O meets the following objective:

- GLE is I/O system-independent. It depends only on having I/O routines of a known format.

Configuration

GLE provides a device-independent interface for all procedures except the initialization and configuration procedures. These are device specific in that the association between the I/O system and GLE is determined in them. The following figure shows the program flow for initialization and configuration.



The first section, Level 1, is composed of the user's graphics language. It is device-independent. The device identifier and device address are passed through this level as part of the GCB in a string called DEVICE_INFO. The contents of the string may be very device- and operating system-dependent, but the string itself is device-independent.

The second section, Level 2 (which is still above GLE), is device- and operating system-dependent. This section is responsible for:

- Determining which devices it should try to initialize,
- Allocating any device-dependent areas (adding to the GCB device-dependent space), and
- Setting up the initialization routine.

In addition, if any device-dependent information is required by the device, such as the address of the start of the frame buffer, then this information must be supplied by the configuration process.

The last section, Level 3, is part of the device handler. It will try to initialize the device and will report an error if it cannot.

When a new device is added to GLE, the following things must be done:

- Write the device handler,
- Modify the configuration module, and
- Write any new I/O routines necessary.
- Higher-level DGL code may also require modification (especially `DGL_CONFG_OUT`).

Buffering

The device handler driver has the option of using buffering. Commands given to the device handler may be converted to device commands and saved in a buffer. The buffer is flushed, or sent to the physical device, either when the buffer becomes full (this must be detected by the device handler), when the procedure `GLE_FLUSH_BUFFER` is called, or after every call to GLE when the buffering mode is set to non-buffered. The HP-IB driver in DGL uses buffering.

Example GLE Program

Following is a program which illustrates several aspects of using the GCBs and other general GLE concepts. The program allows plotting to the CRT, then redirects the plotting commands to go to an area of memory and replots there, then directs the plotting commands back to the CRT. The plotting to memory has the advantage of permitting user-defined frame buffer size. You can have as large a frame buffer as you want, providing you have memory enough to contain the "screen."

Caveats to note about this program:

- It is incomplete. There are variables which are are not dealt with because they are not needed *in this case*, but in a more general case, they would need to be dealt with.
- One specific manifestation of the previous point is that this is a simulation of only a Model 236A frame buffer; different specifications are necessary for simulation of other computers' frame buffers.
- This program uses the heap. Pascal 3.0 DGL tries to stay away from the heap as much as possible, for reasons noted previously.
- There are other ways to do what this program is doing; it doesn't have to be done in the manner described below.

This program is on your Examples disc under the name PLOTTOMEM.TEXT.

```
$sysprog, ucsd$

PROGRAM DUMP_LARGE_GRAPHICS ( INPUT,OUTPUT,LISTING);


import dgl_types,
       dgl_lib,
       dgl_vars,
       dgl_gen,
       gle_types,
       gle_gen,
       gle_ras_out,
       sysglobals,
       asm;

type
  graphics_screen = packed array [1..maxint] of char;

var
  graphics_base ['GRAPHICSBASE'] : anyptr;
  gscreen : ^graphics_screen;
  error_return : integer;
  gscreen_width : integer;
  gscreen_height : integer;
  justify_bytes  : integer;
  marked_screen : anyptr;
  old_graphics_base : ^gle_shortint;
  old_gle_gcb : ^graphics_control_block;
  old_gcb     : ^graphics_control_block1;
  old_raster_gcb,
  tmp_raster_gcb: raster_device_rec_ptr;

function make_anyptr( p : anyptr ) : anyptr;
{    This function converts a pointer of any type into an ANYPTR.  It is     }
{  for assigning values to variables of type ANYPTR.                         }

begin
  make_anyptr := p;
end;

procedure memory_clear(agcb : graphics_control_block_ptr);
{    This is a screen-clear routine which knows how to clear the simulated   }
{  "screen" in mainframe RAM.  The routine is needed because when you first  }
{  do DISPLAY_INIT in the program, a screen clear for your physical display  }
{  is attached to the GLE_GCB hook for "clear".   For some configurations,   }
{  this routine would also serve to clear the simulated screen, but for      }
{  other than Model 236A-equivalent displays, the procedure installed at     }
{  "clear" may not clear the simulated screen properly.                      }

var
 i,j  : gle_shortint;
 row  : integer;
begin
 for i := 0 to gscreen_height-1 do
  begin
   row := i * gscreen_width;
   for j:= 1 to gscreen_width do
```

```
    gscreen^[row + j] := #0;
  end;
end;

procedure take_graphics ( screen_width_dots, screen_height_dots,
                          justify_dots : integer);
{    This procedure puts the currently active display temporarily on hold,   }
{  allocates enough memory to simulate a frame buffer of the size specified  }
{  by SCREEN_WIDTH_DOTS and SCREEN_HEIGHT_DOTS, and redirects plotting       }
{  operations to that memory.                                                }

var
  index : integer;
  raster_gcb : raster_device_rec_ptr;

begin

  mark(marked_screen);

  { save screen size information in global variables }
  gscreen_width := screen_width_dots div 8;
  gscreen_height := screen_height_dots;
  justify_bytes := justify_dots div 8;

  { redirect graphics library variables to point to new memory, }
  { save old values                                             }

  { allocate memory for screen image }
  newwords(gscreen,(gscreen_height*gscreen_width) div 2 + 1);

  new(old_gle_gcb);
  old_gle_gcb^ := gle_gcb^;

  new(old_gcb);
  old_gcb^ := gcb^;

  new(old_raster_gcb);
  tmp_raster_gcb := gle_gcb^.dev_dep_stuff;
  old_raster_gcb^ := tmp_raster_gcb^;

  old_graphics_base := make_anyptr(graphics_base);
  graphics_base := make_anyptr(gscreen);

  raster_gcb := gle_gcb^.dev_dep_stuff;
  with gle_gcb^,raster_gcb^ do
    begin
                                    { \  Take over the screen-clearing hook }
      clear := memory_clear;        {  > to ensure that the "screen" in     }
                                    { /  memory can be cleared correctly.   }

      display_name := 'MEMORY';     { Plotting device is now memory.        }
      display_name_char_count := 6; { Number of characters in 'MEMORY'.     }

                                    { \   Simulate a display whose          }
      display_res_x := 3.0000;      {  \  resolution is 3 pixels/mm.  This  }
      display_res_y := 3.0000;      {  /  is in the ball park for printers  }
                                    { /   which can dump graphics images.   }

      display_min_x := 0;
```

```
display_max_x := ((gscreen_width) * 8 - 1);
display_min_y := 0;
display_max_y := (gscreen_height-1);

color_map_support := 0;  { none }

redef_background := 0;   { no }

pallette     := 1;
gamut        := 1;

devicetype     := 1;  { simulate 9836A only. Other values are
            0 = 16/26, 1 = 36, 2 = 98627A, 3 = 36C, 4 = 9837 }

deviceaddress := 0;   { unused }

plane1_addr := addr(graphics_base);
plane1_offset := 0;
plane2_offset := 0;  {set to 32768 when HP98627A is initialized}
plane3_offset := 0;  {set to 65536 when HP98627A is initialized}

n_glines := display_max_y+1;

                         { \   This specifies the spacing (memory-wise) }
                         {  \  of bytes which affect the frame buffer.  }
gspacing     := 1;       {   > The Models 16 and 26 used only odd bytes }
                         {  /  so the spacing is every two; all others  }
                         { /   use a spacing of one.                    }

bytesperline := gscreen_width;

hard_xmax := display_max_x;
hard_ymax := display_max_y;

with gcb^ do
  begin
    max_disp_lim.xmin := display_min_x;
    max_disp_lim.xmax := display_max_x;
    max_disp_lim.ymin := display_min_y;
    max_disp_lim.ymax := display_max_y;

    gle_get_p1p2 ( gle_gcb );

    def_disp_lim.xmin := info1;
    def_disp_lim.xmax := info2;
    def_disp_lim.ymin := info3;
    def_disp_lim.ymax := info4;

    disp_init := true;
    disp_eq_loc :=  ((disp_dev_adr = loc_dev_adr) or
                    ((disp_dev_adr = internal_display) and
                     (loc_dev_adr = internal_locator)));

    { set up display limits }

                               { \  Is the virtual display coordinate   }
    disp_just := lowerleft;    {  > system centered within the display  }
                               { /  limits or in the lower left corner? }
    with def_disp_lim do
```

```
            display_limits(xmin,xmax,ymin,ymax);

          { set up default text size and rotation attributes }

          dgl_char_width := init_char_width_factor *
            abs (window_lim.xmax - window_lim.xmin);
          dgl_char_height := init_char_height_factor *
            abs (window_lim.ymax - window_lim.ymin);
          set_char_size ( dgl_char_width, dgl_char_height );

          char_rot_w := init_char_rot_w;
          char_rot_h := init_char_rot_h;

          set_text_rot ( char_rot_w, char_rot_h );

          { set up all attributes here        }

          dgl_current_polygon_edge := true;
          dgl_current_polygon_crosshatch := false;
          dgl_current_polygon_linestyle := init_linestyle;
          dgl_current_polygon_style := 1;
          dgl_current_polygon_color := init_color;
          dgl_polygon_color_current := false;  { color not set in gle }
          dgl_current_polygon_density := 0;
          dgl_current_polygon_angle := 0;
          set_timing ( dgl_current_timming_mode );
          set_color(init_color);
          set_line_style(init_linestyle);
          set_line_width(init_linewidth);

          cpx := init_cpx;      { \   Set the current pen position to the  }
          cpy := init_cpy;      {  >  initial current pen position.  The   }
                                { /   units are device units.              }

          marker_size_x := trunc(display_res_x * 2.5 + 0.5); { 2.5 mm in size }
          marker_size_y := marker_size_x;
          info1 := marker_size_x;
          info2 := marker_size_y;
          gle_marker_size ( gle_gcb );
        end;
    end;
    clear_display;
end; { setup_display }

procedure return_graphics;
{    This procedure performs the inverse function of TAKE_GRAPHICS.  It     }
{  redirects plotting operations back to the display and away from the      }
{  pseudo-display in memory.  It also destroys the pseudo-display           }
{  information and releases the pseudo-frame buffer from the heap.          }

var
  error : integer;

begin
  graphics_base := make_anyptr(old_graphics_base);

  gle_gcb^ := old_gle_gcb^;
  gcb^     := old_gcb^;
  tmp_raster_gcb := gle_gcb^.dev_dep_stuff;
```

```
  tmp_raster_gcb^ := old_raster_gcb^;

  release(marked_screen);

  with gcb^.def_disp_lim do
   display_limits(xmin, xmax, ymin, ymax);

end;

procedure dump_graphics;
{    This procedure dumps the graphics image in the memory to a printer which }
{  can do a graphics dump.  The printer must conform to the HP Raster         }
{  Interface Standard in order to work with this procedure.                   }
{    The memory-display must be less than 132 characters (1056 pixels)      }
{  wide.                                                                      }

label 1;

var
  gbuffer : string[138 { 132 + 6 }];
  i,j,pindex : integer;
  busy : boolean;
  row : integer;
  cnt : integer;

begin
1:
  { escape sequence for graphics }
  gbuffer := '';
  strwrite(gbuffer,1,cnt,chr(27),'*b',gscreen_width:0,'W');
  cnt := cnt - 1;
  setstrlen(gbuffer,gscreen_width+cnt+justify_bytes);

  try
    for i := 1 to justify_bytes do
      gbuffer[cnt+i] := chr(0);

    for i := 1 to gscreen_height do
      begin
        row := (i - 1) * gscreen_width;
        for j := 1 to gscreen_width do
          begin
            gbuffer[j+cnt+justify_bytes] := gscreen^[row+j];
          end;
          WRITE(LISTING,GBUFFER:GSCREEN_WIDTH+6);
      end;
  recover ;

  gbuffer[1] := chr(27); { terminate graphics sequence }
  gbuffer[2] := '*';
  gbuffer[3] := 'r';
  gbuffer[4] := 'B';
  WRITE(LISTING,GBUFFER:4);

end;
procedure pattern(xmin,xmax,ymin,ymax: real);
{    This merely draw a pattern on the display (or pseudo-display) to prove  }
{  that the hooks needed for plotting have been correctly assigned.          }
```

```
const
  convert_deg_to_rad = 0.01745329252;

var
  dx,dy : real;
  deg   : integer;
  cnt   : integer;
  s : string[20];

begin
  dx := xmax-xmin;
  dy := ymax-ymin;
  set_window(xmin,xmax,ymin,ymax);
  set_aspect(dx,dy);
  move(xmin,ymin);
  line(xmin,ymax);
  line(xmax,ymax);
  line(xmax,ymin);
  line(xmin,ymin);
  set_char_size(dx/25,dy/25);
  deg := 0;
  repeat
    move(dx/2,dy/2);
    set_text_rot(cos(deg*convert_deg_to_rad),sin(deg*convert_deg_to_rad));
    s := '   ---- ';
    strwrite(s,7,cnt,deg:1);
    gtext(s);
    deg := deg + 25;
  until deg > 340;
end;


begin
  graphics_init;
  display_init ( 3,0,error_return);
  if error_return <> 0 then escape(-27);
  pattern(0,1,0,1);
  set_line_style(3);            {set a non-default line style so if something }
                                {goes wrong, it will be obvious.              }
  take_graphics(560,720,0);
  pattern(0,0.5,0,0.5);
{ dump_graphics;                {un-comment this if you have a printer }
                                {which can dump graphics.             }
  return_graphics;

  move(0.25,0.25);      {draw a line in line style 3 on the CRT.  If it }
  line(0.75,0.75);      {is not a dashed line, something went wrong.    }

  graphics_term;

end.
```

Drivers

In the context of DGL and/or GLE, a driver is a suite of procedures whose purpose is to communicate with a specific class of devices. DGL (including GLE here) contains only two drivers:

- The raster display driver. The procedures related to this are contained in the modules GLE_RAS_OUT and GLE_ARAS_OUT.
- The HPGL input/output driver. The procedures related to this are contained in the modules GLE_HPGL_IN and GLE_HPGL_OUT.

Functional Description

A driver, whether supplied with the system, or one that you create for a specific purpose, must take care of all the communication required to cause the desired device to carry out every function that is desired by the software designer. Some devices do not need every capability defined that another device has. For example, a hard-copy plotter driver need not respond to a command to "clear the screen;" it is nonsensical on a plotter that cannot move its own paper.

The GLE drivers specify the routines which are needed carry out the desired functions in a way that the specified hardware requires. The proper procedures are indicated by assigning the values of procedure variables in the graphics control blocks to the appropriate routines. This is done as one of the consequences of calling DISPLAY_INIT (GRAPHICS_INIT does not assign drivers).

For instance, in the procedure GLE_INIT_HPGL_OUT in the module GLE_HPGL_OUT, the procedure variables are assigned the values of procedures which cause HPGL operations to take place. As you may remember from the GLE GCB listing given earlier in the chapter, the procedure variables are fields in the GCB. Note in the following segment of code (taken from procedure GLE_INIT_HPGL_OUT) that the procedure variables are set to appropriate values for HPGL device.

```
begin
  with gcb^,
       hpgl_device_rec_ptr(dev_dep_stuff)^,
       ascii_buffer_ptr(device_buf)^ do
    try
      current := 0;
      maximum := max_buffer;

      driver_state          := start_of_buffer;
      unclipped_move        := hpgl_move;
      unclipped_draw        := hpgl_draw;
      move                  := gle_soft_clip_move;
      draw                  := gle_soft_clip_draw;
      clear                 := hpgl_clear;
      text                  := gle_soft_text {hpgl_text};
      char_size             := gle_soft_char_size;
      clip_limits           := gle_soft_clip_limits;
      text_spacing          := gle_soft_text_spacing;
      linestyle             := hpgl_linestyle;
      text_dir              := gle_soft_text_dir;
      text_just             := gle_soft_text_just;
      marker                := gle_soft_marker;
      marker_size           := gle_soft_marker_size;
      set_marker            := gle_soft_set_marker;
```

```
index_color          := hpgl_set_color;
inq_p1p2             := hpgl_get_p1p2;
get_polygon_info     := dummy;
graphics_on_off      := dummy;
cursor               := hpgl_cursor;
calc_soft_text_xform := gle_text_xform;
buffer_mode          := hpgl_buffer_mode;
output_escapeo       := hpgl_output_escapeo;
output_escapei       := hpgl_output_escapei;
define_drawing_mode  := dummy;
define_color_map     := dummy;
polygon              := dummy;
fill_index_color     := dummy;
linewidth            := dummy;
gload                := dummy;
gstore               := dummy;
get_raster           := dummy;
get_color_map        := dummy;
await_blanking       := dummy;
flush_buffer         := hpgl_flush_buffer;
    .
    .
    .
```

Then, when a higher-level routine wants to draw a line on an HPGL plotter, it merely says:

```
call(gcb^.draw, gcb);
```

The routine that gets called is the routine pointed at by the value of `GCB^.UNCLIPPED_DRAW`, which, after the above code is executed, is the procedure `HPGL_DRAW`.

Note that the routines which are unnecessary are not left undefined, but are assigned to a "dummy" procedure, usually consisting of an empty `BEGIN/END` pair. Note also that some of the procedure variables can be assigned to tools (e.g., `MOVE:=GLE_SOFT_CLIP_MOVE;`).

Syntactical Description

Most of the procedure variables have a single parameter: the GCB. To keep your code compatible, you must make your routines similarly, for example:

```
procedure custom_made_draw(gcb: graphics_control_block_ptr);
```

Note that the GCB is a pointer passed by value, not by reference. Passing a pointer by value is roughly equivalent to passing a variable by reference.

Driver Data Structures

When creating a driver for a certain class of devices, you will probably want to define a data structure which can conveniently pass or retain any necessary values for the driver routines. This is what HP did when writing the raster and HPGL drivers.

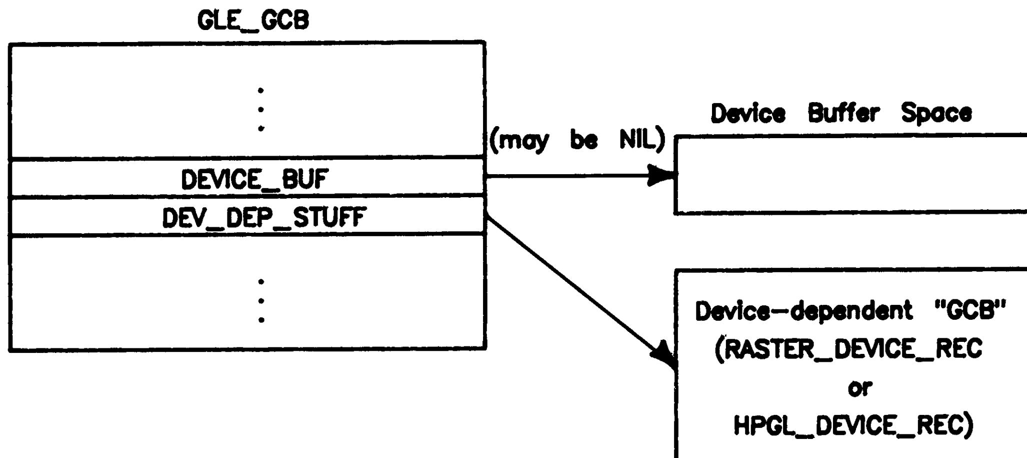
Raster Driver Data Structure

The data structure for the raster-related driver is called RASTER_DEVICE_REC (also referred to as RGL_GCB in some assembly routines), and it is exported from module GLE_RAS_OUT. It is pointed to from GLE_GCB, and was defined earlier in this chapter under the heading *GLE Initialization*, subheading *GLE_RAS_OUT*.

HPGL Driver Data Structure

The data structure for the HPGL-related driver is much simpler than that of the raster driver. Called HPGL_DEVICE_REC, it is exported from module GLE_HPGL_OUT. It is defined as follows:

```
type
  driver_state_def = (moving,drawing,start_of_buffer,unknown);

  hpgl_device_rec_ptr= ^ hpgl_device_rec;
  hpgl_device_rec =
    record
      driver_state : driver_state_def;
    end;
```



HPGL Move Example

Below is an example of the operations the DGL routines go through in order to cause a graphics MOVE operation. This assumes that the graphics library has already been initialized (GRAPHICS_INIT), and the "display"—actually an HPGL plotter—initialized as well (DISPLAY_INIT).

1. The user program executes the library procedure MOVE, which is exported from module DGL_LIB.
2. The DGL_LIB procedure MOVE scales X and Y, sets up some DGL variables, sets GLE_GCB^.END_X and GLE_GCB^.END_Y to the end point in device units, and calls the procedure pointed to by the procedure variableGLE_GCB^.MOVE. The value of the procedure variable GLE_GCB^.MOVE is a procedure called GLE_SOFT_CLIP_MOVE, exported from GLE_ASCLIP. This assignment to GLE_GCB^.MOVE was specified by procedure GLE_INIT_HPGL_OUTPUT, which was exported from GLE_HPGL_OUT.
3. After clipping, the assembly language procedure GLE_SOFT_CLIP_MOVE calls the procedure pointed to by GLE_GCB^.UNCLIPPED_MOVE, whose current value is HPGL_MOVE.
4. The procedure HPGL_MOVE prepares to place characters into a buffer. This process is optimized by the use of a variable called DRIVER_STATE, an enumerated type defined to be (MOVING, DRAWING, START_OF_BUFFER, UNKNOWN). If HPGL_MOVE sees that the current state of the driver is MOVING, the "start moving" command need not be sent; it need only append the new coordinates to which to move (after appending a comma to the buffer). If the current state of the driver is not MOVING, only then is the HPGL "start moving" command placed in the buffer. This command consists of the following:
   a. Terminate the previous HPGL command by sending a semicolon: ';'.
   b. Tell the plotter to pick the pen up: 'PU;'.
   c. With the pen up, start the plot absolute command: 'PA'.
5. After ensuring that the state of the driver is MOVING, the actual characters specifying the destination location are placed into the buffer:
   a. Procedure ADD_PARM_DATA, defined in GLE_HPGL_OUT. This converts the number passed to it to ASCII characters (via GLE_WRITE_INTEGER, exported from GLE_UTLS), and then appends the ASCII version of the number to the end of the buffer, updating CURRENT.
   b. Procedure ADD_CHAR_DATA, defined in GLE_HPGL_OUT. This appends HPGL instructions to the end of the buffer.
   c. Updates the current X and Y position of the pen.
   d. Calls procedure BUFFER_CLEANUP, which is defined in GLE_HPGL_OUT.
6. Procedure BUFFER_CLEANUP may call HPGL_FLUSH_BUFFER (also defined in GLE_HPGL_OUT).
7. If HPGL_FLUSH_BUFFER is called and if the buffer contains data (and it does; the MOVE just put it there), the procedure HPGL_FLUSH_BUFFER calls the procedure pointed to by the procedure variable IO_WRITE, which was assigned the value of HPIB_WRITE by the procedure SETUPHPGL in module DGL_CONFG_OUT.

8. Procedure HPIB_WRITE, exported from GLE_HPIB_io, calls the system drivers which output the buffer contents to the plotter. This takes place thus:
   a. Executes the function ADDR_TO_LISTEN to get the HP-IB addressed so the plotter is listening, returning the select code part of the device selector to IO_ISC. For example, ADDR_TO_LISTEN(705) sets bus address 5 to listen on select code 7 (if 7 is an HP-IB), and then returns 7 as the value of the function.
   b. It invokes the procedure pointed to by the procedure variable IOD_WTB to send each character in the buffer (WTB stands for "write byte"). This is defined in the module IODECLARATIONS.
   c. Then, WRITECHAR (in GLE_HPIB) is called twice, to send carriage return/linefeed.
   d. The buffer is set to empty (CURRENT:=0;).
9. Upon returning from HPIB_WRITE, BUFFER_CLEANUP sets DRIVER_STATE to START_OF_BUFFER.
10. The plotter moves the pen.

Graphics System Initialization

There are two main user-level procedures involved with the initialization of the graphics system. They initialize procedure variables, numeric variables, strings, and, to indicate that the initialization has taken place, booleans are set to true.

The two procedures we will cover in this section are:

- GRAPHICS_INIT,
- DISPLAY_INIT (in this case, setting up an HPGL device),

GRAPHICS_INIT

The basic program flow for the procedure GRAPHICS_INIT is as follows:

1. If the graphics system is already initialized, then terminate it. If display and locator are initialized, this will terminate them, possibly flushing the buffer, and thus sending some final commands to a device.
2. Set SYSTEM_INIT to TRUE, indicating that the graphics system, as a whole, will have been initialized by the time anyone can check this boolean. Two other booleans, DISP_INIT and LOC_INIT, are set to FALSE to indicate that a neither a display nor a locator has been initialized yet.
3. Set up pointers to the storage areas containing the GCBs and other global variables. Storage areas include space for GCB, GLE_GCB, GLE_GCBI, and GLE_KNOB_ECHO_GCB. (Remember, Pascal 2.x DGL used dynamic allocation for this; Pascal 3.0 DGL uses globals.) Initialize the GLE GCBs to "uninitialized."
4. Notes if a Katakana keyboard is installed. Make a note of the answer in GLE_GCB^.KATA.
5. Sets dozens of values in the DGL GCB to their defaults. The segment of code below enumerates them.

```
{ set up defaults }
with gcb^ do
  begin
    { When first initialized, the display and locator are the same device }
    disp_eq_loc := true;
    disp_dev_adr := init_dev_adr;
    disp_file_name := '';
    loc_dev_adr  := init_dev_adr;
    window_lim   := init_window;
    aspect_ratio := init_aspect;
    cur_vir_lim  := init_vir_lim;
    viewport_lim := init_viewport;

    { setup the default display/ locator limits to some large number }

    with init_display_lim do
      display_limits ( xmin, xmax, ymin, ymax );
    with init_locator_lim do
      locator_limits ( xmin, xmax, ymin, ymax );

    { explicitly set the cp to init_value }
```

```
    cpx := init_cpx;
    cpy := init_cpy;
    int_cp := true;
    world_int_cpx := init_cpx;
    world_int_cpy := init_cpy;

    { set up default text size and rotation attributes }

    dgl_char_width := init_char_width;
    dgl_char_height := init_char_height;

    char_rot_w := init_char_rot_w;
    char_rot_h := init_char_rot_h;

    { set flag, indicating that the text xform needs to be recalculated      }

    calc_text_xform := true;

    dgl_current_color := init_color;
    dgl_current_linestyle := init_linestyle;
    dgl_current_linewidth := init_linewidth;
    dgl_current_timming_mode := init_timming_mode;
    cursor_color := init_color;
    disp_just := centered;
    display_echo_mult := 1;
    graphics_error := 0;

    number_polygon_styles := default_poly_table_size;
    color_table_size := default_color_table_size;
    dgl_current_polygon_color := 1;
    dgl_current_polygon_linestyle := 1;
    dgl_current_polygon_density := 0;
    dgl_current_polygon_angle := 90;
    dgl_current_polygon_edge := true;
    dgl_current_polygon_crosshatch := false;
    dgl_current_polygon_style := 1;
    dgl_current_color_model := 1;
  end;
```

See the code listings for further details.

DISPLAY_INIT

In this section, we will follow the logic flow through the routine `DISPLAY_INIT`, having specified that we are initializing for an HPGL device.

1. Call `CK_SYSTEM_INIT` to see if the graphics system is initialized. The procedure is exported from `DGL_GEN`, and is a very simple routine:

```
procedure ck_system_init;

{ Purpose : To report an error if the system is not initialized        }

begin
  if not system_init then error (err_sys_int);
end; { ck_system_init }
```

2. Check to see if a display is already initialized. If so, call DISPLAY_TERM to terminate the previously-used display.

3. Initialize several GLE variables:

```
with gle_gcb^ do
    begin
      s := '';
      strwrite(s,1,cnt,dev_adr:0);
      device_info_char_count := strlen(s);
      device_info := addr(s[1]);
      spooling := 0;
      info1 := control;
      info2 := 0; { config DGL stuff }
      configure_gle (gle_gcb);
      ierr := error_return;
    end;
```

4. The CONFIGURE_GLE procedure does the following:
   a. Try to initialize a raster display (for a blow-by-blow account of raster initialization, see the section *Raster Display Initialization* later in this chapter.
   b. If the raster initialization attempt succeeded, quit.
   c. If the raster initialization attempt failed, try to initialize an HPGL device.
   d. If the HPGL initialization attempt succeeded, quit.
   e. If the HPGL initialization attempt failed, set error flag ERROR_RETURN and return to the calling routine.

5. If the previous initialization process (CONFIGURE_GLE) was successful, the variable ERROR_RETURN, and therefore IERR, will be zero. If it is zero, then define the two GCB fields DISP_DEV_ADR (display device address) and DISP_FILE_NAME (display file name). Then, call SETUP_DISPLAY, which does the following:
   a. Sets the GLE GCB X and Y *maximum* display limits and *default* display limits.
   b. Set DISP_INIT to TRUE: Yes, the display is initialized.
   c. Sets a boolean that indicates whether or not the display device is that same as the locator device.
   d. Set various text-related values: DGL_CHAR_WIDTH, DGL_CHAR_HEIGHT, CHAR_ROT_H, and CHAR_ROT_W.
   e. Initialize some more variables:

```
dgl_current_polygon_edge := true;
dgl_current_polygon_crosshatch := false;
dgl_current_polygon_linestyle := init_linestyle;
dgl_current_polygon_style := 1;
dgl_current_polygon_color := init_color;
dgl_polygon_color_current := false;  { color not set in gle }
dgl_current_polygon_density := 0;
dgl_current_polygon_angle := 0;
set_timing ( dgl_current_timming_mode );
dgl_current_color := -1; { force calc of color }
set_color(init_color);
set_line_style(init_linestyle);
set_line_width(init_linewidth);
```

```
{ init_cpy is in device units }
cpx := init_cpx;
cpy := init_cpy;
```

f. Define marker-related information.

g. Make sure the graphics display is on.

6. If the display supports complement mode (e.g., a raster device), set GLE_GCB^.CURSOR to DGL_CURSOR in DGL_LIB. DGL_CURSOR assumes the routine calling it has already set up complement mode, so that to erase display information, it need only redraw the same lines.

Configuring Your Own Driver

The routine CONFIGURE_GLE, mentioned above, quits after having failed on a raster device initialization and an HPGL initialization:

```
procedure configure_gle (  gcb : graphics_control_block_ptr );

VAR
    save : integer;

begin
  with gcb^ do
    begin
      setupraster ( gcb );
      if error_return <> 0 then setuphpgl ( gcb );
    end;
end;
```

To configure your own driver, you would just extend the logic a little more:

```
procedure configure_gle (  gcb : graphics_control_block_ptr );

VAR
    save : integer;

begin
  with gcb^ do
    begin
      setupraster ( gcb );
      if error_return <> 0 then setuphpgl ( gcb );
      if error_return <> 0 then setup_custom_driver1 ( gcb );
      if error_return <> 0 then setup_custom_driver2 ( gcb );
    end;
end;
```

As many custom-made drivers as desired can be configured this way.

Raster Display Initialization

As briefly mentioned before in the discussion on output device configuration, CONFIGURE_GLE attempts to set up a raster display by calling the procedure SETUPRASTER, in module DGL_CONFG_OUT. This section details a more in-depth description of the workings of SETUPRASTER. First, a small glossary is presented, defining some of the terms which will be used in the description itself. After that comes the description of the configuration process, which covers the high points of the operation of SETUPRASTER, but is not meant to be exhaustive. See the code listings for that.

"Alpha" Display	A CRT which has an alpha raster and (optionally) a graphics raster which can be turned on and off independently. They may or may not have the same resolution, and operations done to one raster do not affect the other raster.
"Bit-mapped" Display	A bit-mapped display is a CRT which has only one raster—both alpha and graphics operations send information to the same area. After a pixel has been turned on, there is no way to tell whether an alpha operation or a graphics operation activated the pixel.
Hi-Res Display	A Model 237 display. This display is a bit-mapped monochrome display and has a resolution of 1024×768.
ID Register	The ID register is a byte at $560001 for internal displays or $600000+$10000×*(select code)*+$1 for external displays.
Lo-Res Display	This is any display except the Model 237 display. These are all alpha-type displays.
Primary Display	The display to which the file system sends all alpha output unless specifically directed elsewhere; this is where the Pascal command line appears. For DGL graphical output to be directed to the primary display, the token *3* must be sent to DISPLAY_INIT.
Secondary Display	Any display other than the console that does not require a select code and/or bus address to access it. For DGL graphical output to be directed to the secondary display, the token *6* must be sent to DISPLAY_INIT.
Spooling	Placing output information onto a file as a temporary holding place. Usually, some automatic program (a "spooler") periodically checks for spooled files, and does the actual transfer of the data to the output device. To cause DGL output to be spooled, call DISPLAY_FINIT, as opposed to DISPLAY_INIT. There are no supported "spoolers" on the Pascal Workstation. However, the SRM (Shared Resource Management) version 2.0 does do primitive plotter spooling.

Initialization Description

1. Check to see if the output is to be spooled. Since raster devices' output cannot be spooled, the procedure will give an error if you are spooling (i.e., executing `DISPLAY_FINIT`). `ERROR_RETURN` will be set to 1 and the routine will quit.

2. Attempt to locate a Model 237 high-resolution (1024×768) display. This is accomplished by calling routine `GATORCRTTYPE`, also in module `DGL_CONFG_OUT`.

   a. It checks the value of the ID register at `$560000`, masking off bit 7 (the remote bit). If certain bits are set, an internal high-res display exists; set `INT_EXT_GATOR:=1`. If a CPU bus error occurred (the ID register was not present), an internal display was not found.

   b. If an *internal* hi-res display was not found, an *external* hi-res display may exist. Now check the value of the ID register at `$680000`. If certain bits are set, an external high-res display exists; set `INT_EXT_GATOR:=2`. If a CPU bus error occurred (the ID register was not present), an external display was not found.

   c. If either an internal or external hi-res display was found, a status pointer is set up as well as a pointer to the frame buffer. If no hi-res display was found, the procedure exits with `FOUND_GATOR` set to `FALSE`, and `INT_EXT_GATOR` set to 0.

3. Check for the existence of low-resolution, built-in graphics hardware in the machine as part of the alpha display. A nested procedure, `CK_FOR_GRAPHICS_BOARD`, in the procedure `SETUPRASTER` does the checking. It attempts to read a word from the area in memory where the frame buffer would be, and if no CPU bus error occurs, the hardware exists.

4. Assign values to several variables in the GLE GCB, including a pointer to the `RGL_GCB`.

5. Check to see if the address of the desired display is the token *3*, indicating the primary display, the same place (if possible) as is used for the alpha display. If so, do the following:

   a. If low-resolution graphics hardware is in the machine, and either the current console alpha display is an alpha-type (non-bit-mapped) CRT or no internal CRT exists at all, set up the internal (lo-res) display. This includes:

      - Determine the type of machine running the software.
      - Turn the graphics hardware on if the state variable `GRAPHICSTATE` is true. Otherwise, turn it off.
      - Set a few variables.

   b. If the condition in step *5a* is not true, then check: do we have an internal hi-res bit-mapped display and either (1) the current CRT is bit-mapped or (2) we do not have internal lo-res graphics hardware? If so, then set up for a hi-res display:

      - Define the display type.
      - Set a few pointers and control registers.

6. If the condition in step *5* is false, then check to see if the address of the desired display is the token *6*, indicating the secondary display. If so, do the following:

   a. If the current alpha CRT is an alpha-type display, or no internal CRT exists, and a hi-res display exists, set it up. Otherwise (if the hardware is there, of course), set up the internal display.

b. If the current alpha CRT is a hi-res (bit-mapped) display, but graphics hardware for an internal lo-res display exists, set the secondary display to the lo-res display. If the current CRT is a hi-res (bit-mapped) display, but graphics hardware for an internal lo-res display *does not* exist, then set up to use the hi-res display.

c. If the conditions in both *6a* and *6b* are false, then check: if the specified address is an external select code (8 through 31), and an external hi-res display exists, then set up for a hi-res display. Otherwise, try to set up for an HP 98627A external color monitor.

7. Deal with the control word. This specifies whether or not to "expand the screen." Expanding the screen deals with the hi-res bit-mapped display. Since the alpha screen can be the same as the graphics screen on a bit-mapped display, the possibility exists that a graphical output might interfere with the type-ahead buffer and vice versa. If the requested graphics display screen is the primary display, then the type-ahead buffer is echoed to that screen; therefore use a shrunken version of the graphics screen, smaller by sixteen scan lines, making it 752 pixels high. If the `CONTROL` bit 8 is set in `DISPLAY_INIT`, use the whole 768 scan lines for the display, disabling `CRTLLHOOK` (which writes alpha to the last 16 pixel rows).

8. Call `GLE_INIT_RASTER_OUTPUT`, exported from `GLE_RAS_OUT`, which does more raster display initialization.

9. If the condition in step *8* went well, call `DGL_RASTER_INIT`, exported from `DGL_RASTER`. This sets up display characteristics for the current device, including markers, polygons, line styles, color map information, and procedure variable assignments.

Floating Point Card 13

Introduction

This chapter describes the Pascal 3.0 software which deals with the 98635A floating point card. The information in this chapter contains a description of the design of the software/hardware interface, along with some examples on how to program and debug the card from assembly language. It also contains a brief summary of some of the changes necessary to support the 98635A card that were made to the Pascal Workstation environment.

The following are terms used throughout the text.

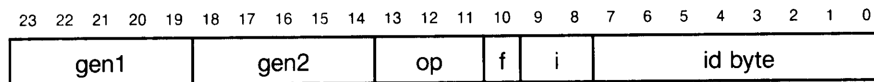
integer	A 32-bit integer
float	A 32-bit IEEE format floating point number
long	A 64-bit IEEE format floating point number
ISR	Interrupt Service Routine
FPU	Floating Point Unit (National's 16081)

All numbers prefixed with a "$" are hexadecimal. The 16081's floating point registers are referenced with the mnemonic "`f`x", where x is in the range 0 through 7.

National's Hardware

The National 16081 FPU is a slave processor in National's NS16000 microcomputer family. It provides a high-speed floating point instruction set for both IEEE 32- and 64-bit format numbers, as well as support for the IEEE proposed standard (P-754) for binary floating point numbers. The set of operations supported by the FPU consists of the four basic arithmetic operations of addition, subtraction, multiplication, and division, as well as general data movement and conversions between integers (8, 16, and 32 bits) and floating point types. The National 16000 family has nine different addressing modes, all of which are applicable to the Floating Point Unit. There are eight 32-bit floating point registers on the chip, as well as a 32-bit status/control register.

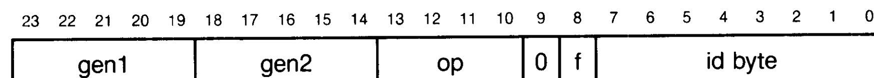
The 16081 instruction set has essentially two formats, described below.



where the fields have the following meanings:

gen1 (5 bits)	Source operand
gen2 (5 bits)	Destination operand
op (3 bits)	Operation to be done
f (1 bit)	Floating point format
i (2 bits)	Integer type
id byte (8 bits)	See below

National's 16081 Format 9 Instruction



where the fields have the following meanings:

gen1 (5 bits)	Source operand
gen2 (5 bits)	Destination operand
op (4 bits)	Operation to be done
0 (1 bit)	Not used
f (1 bit)	Floating point format
id byte (8 bits)	See below

National's 16081 Format 11 Instruction

The `ID BYTE` has three functions. It identifies the instruction to the National CPU as being a slave processor instruction, it specifies which slave processor is to execute the instruction (the 16081 is but one of the slave processors in the National family) and it determines the format of the instruction (i.e., either format 9 or 11).

A brief description of how the FPU works in conjunction with the National 16000 CPU follows.

Upon receiving the slave processor instruction (identified by the `ID BYTE`), the CPU transfers the `ID BYTE` to the FPU, which now becomes activated. From this point on, the CPU is communicating only with it. The CPU now sends the operation word (which is nothing more than the rest of the instruction), and the FPU decodes it. At this point both the CPU and the 16081 are aware of the number of operands and their sizes. Using the addressing modes within the operation word, the CPU starts fetching operands and issuing them to the FPU. After the last operand has been transferred, the FPU starts the actual execution of the instruction. Upon completion, it signals the CPU by pulsing one of its lines low. The FPU then broadcasts a status word that indicates if an error has occurred. If no error has happened, the CPU reads the result (if any) and transfers the data to its destination.

Current FPUs being shipped by National have at least one known bug. If the underflow exception is enabled, an underflow error is signaled when trying to convert a 64-bit real 0 into a 32-bit real 0. Because the Pascal Workstation does not support 32-bit reals, this problem will not show up.

The Pascal Workstation's Design and Interface

The National FPU could not easily be connected directly to the 68xxx CPU due to the fact that is was designed to connect with the National 16000 CPU. However, it appeared that it was possible to use the FPU essentially as a peripheral. By building an IO card with the FPU and communicating with it via memory-mapped registers, it appeared that a reasonable floating point product could be built.

There are several ways of connecting the 68xxx and the National FPU. One of the ways first looked at was having the 68xxx, in conjunction with some additional hardware on the I/O card, emulate the National 16000 CPU. There would be several memory-mapped registers that would control the operation of the FPU; that is, there would be registers to deliver data to the card, registers to initiate FPU processing, and remove data from the card. All possible operations would then be achievable with this scheme. For example, to multiply two longs in memory would require the following sequence of events:

1. The software places the opcode for a memory-to-memory long real multiply in a predetermined memory-mapped address.
2. The I/O card then activates the FPU and generates all necessary data/signals and waits for the longs from the 68xxx.
3. The software sends the data in the appropriate order, 16 bits at a time to another memory-mapped address.
4. When the last 16 bits are delivered, the FPU starts the multiply.
5. The software waits for a signal from the I/O card, signifying the operation is complete.
6. The software takes the result from the I/O card via reads from the appropriate memory-mapped addresses.

The major problem with this design is the fact that the sequence of instructions necessary above is non-interruptible. If an interrupt occurs while the data is being transferred in, the FPU is left in a non-savable state due to the fact that its internal microcode is expecting more data; there is no way to save and restore the FPU's internal microcode. Also, there is no simple way to back out and restart the operation to the card; the ability to restart a sequence of instructions is not inherently a part of the 68xxx architecture.

The only solution to the problem (given the proposed hardware design) would be to make the sequence of above instructions non-interruptible; that is, the instructions must be executed at the system's highest interrupt level.[1] However, this is not possible in the Pascal Workstation, for, among other things, the following reasons:

- Clock operations generate interrupts. To avoid these, you would need to guarantee that no timer interrupts were possible when the FPU was going to be used.

[1] Of course, another solution is not to allow any ISR to use the card to do floating point; this does not work, however, in a multiprocess environment in which several processes may be sharing the card and interrupts cause context switches.

- All keystrokes generate interrupts. To avoid this, the typeahead buffer would need to be deactivated.

Clearly, this design would not be adequate for the Pascal Workstation environment. How then, could a design be incorporated that

- Allowed multiple processes and ISRs,
- Got relatively good performance gains, and
- Had a relatively simple software interface?

The state of a quiescent FPU is entirely described by its eight 32-bit floating point registers, its 32-bit status and control register, and its last broadcast 16-bit status word (in our implementation, the last broadcast status word is mapped into a byte on the card), all of which are easily read from and written to. Therefore, it was imperative to communicate with the FPU with 68xxx instructions that caused the FPU to go from one quiescent state to another. There could not be any *sequence* of instructions that were necessary for the FPU to go from one quiet state to the next; if such a sequence existed, it would be possible for an interrupt to occur during the sequence and leave the FPU in a non-restorable state. Therefore, all communication to and from the FPU must be done via single 68xxx instructions.

However, this is not to say that sequences of instructions are verboten; in many cases the floating point card cannot complete an operation before a bus fault (`DTACK` timeout) occurs; in these cases, some form of 68xxx⇔FPU synchronization via either a waiting or polling mechanism must be implemented. These "wait" virtual instructions are special—they are merely interrogating the FPU, not causing a state change. Therefore, an interrupt can occur directly before a "wait" virtual instruction and the FPU will still be in a savable state. The key is that the FPU is not expecting anything more externally; it merely needs to finish what it is currently doing. For more on this, see the section on *Waiting*.

Given this single instruction directive, how then can a memory-memory long real multiply be accomplished? If the FPU is looked at as a peripheral, there are three basic things that can be done:

- Data can be written to it;
- Data can be read from it; and
- It can be instructed to operate on data.

We can perform the multiply in a different manner if instead the data is written to the FPU's floating point registers, the multiply is performed with the result being in an FPU register, and the result is moved from the FPU into memory. Fortunately, each of these virtual operations is supported quite nicely by the FPU's instruction set and each of these steps can be performed with the proper granularity from the 68xxx to insure that the FPU is in a quiescent state upon an interrupt. How this is accomplished will be discussed in the next section.

Given this floating point model, much of the burden of direct communication with the FPU was placed on the hardware. The hardware was made to do as much as possible in order to make the software interface easy and to squeeze out as much performance as possible. There were several things that the software could have performed, but the hardware was made to do for performance and/or programming purposes. First of all, the hardware generated the ID byte for the instructions. Secondly, the hardware was responsible for reversing the words in the data received by the FPU; Motorola and National have different memory layouts (the low and high 16-bit words of a 32-bit datum are in opposite order). The hardware swapped the 32-bit words coming from the 68xxx and fed them into the FPU in the correct order. Finally, the hardware generated the necessary National FPU instructions from a set of PROMs on the card; it was not necessary for the software to know anything about the format of National 16000 instructions.

Some simplifications in the design were made to either improve performance or cut down on the cost of the hardware. The first simplification resulted from the fact that only one of National's memory addressing modes was used in the operand fields in the FPU instructions. This was possible due to the fact that the FPU really doesn't care about what *type* of memory addressing modes are used; it needs to know only that its source/destination operands are coming from or going to memory because the CPU does all of the address calculations. Another simplification resulted from the decision to support only 32-bit integers, the main reason being the easy ability (with the 68xxx "`EXT`" instruction) to convert 8- and 16-bit integers to 32 bits. Supporting only certain sets of floating point operations (as opposed to supporting the entire FPU capability set) based on their frequency of occurrence in typical programs and how efficient the current software was in handling them also resulted in some hardware and software savings that did not compromise either performance or functionality.

Talking to the Card

The model of the hardware as viewed from the software is a set of eight floating point registers (register pairs are used for longs) and a 32-bit status/control register that can be read from or written to via a set of memory-mapped addresses. There also exists a set of memory-mapped addresses used to perform floating point operations on the data in these registers.

Writing Data to the Card

All data is written in 32-bit quantities to a set of memory-mapped addresses that transfers the data (with conversions, if appropriate) to one of the eight FPU floating point registers or to the FPU status/control register. When data is written to a specific location, a hardware state machine on the card retrieves the correct National instruction from PROMs that contain the set of instructions used in our implementation, and generates all the necessary signals to place the data into one of the FPU registers. An example will serve to illustrate.

Suppose it was desired to place a float that was in 68xxx register `D0` into FPU register `f0`. The code sequence

```
move.l  d0,⟨addr⟩    ⟨addr⟩ is a memory-mapped card register
```

would write the float to the floating point card, which then retrieves the National FPU instruction to place a float from memory into `f0`[2]. The card then feeds the FPU this instruction, followed by the data. The instruction in the PROM retrieved by the card is (in National mnemonics) "`MOVF` ⟨*memory*⟩`,f0`"; this National Format 11 instruction is: Move (without conversion) a 32-bit floating-point quantity from memory (which in this case happens to be a 68xxx data register) into the FPU register `f0`.

It is very important that this operation complete before a bus fault (`DTACK` timeout) occurs, mainly so that a "wait" virtual instruction does not need to used for every move of data into the FPU. Fortunately, the floating point card can do almost all of these type of operations without any form of waiting, except for moves of integers into the FPU registers while converting them into either floats or longs. In this case, the FPU cannot do the conversion fast enough; therefore, a "wait" virtual instruction must be placed after writing to the card. If this "wait" is not present and another 68xxx instruction tries to do something with the card before the FPU and the card are quiescent, data corruption is guaranteed to happen.

The operations of writing data to the card consist of:

- Moving floats and longs into the FPU,
- Moving floats into the FPU, converting them into longs,
- Moving integers into the FPU, converting them into longs and/or floats, and
- Setting the 32-bit status/control register.

Notice also how this class of instructions fits in nicely with expression evaluation. Typically, expression evaluation involves converting operands up to the most complex type and then doing the evaluation. This class of operations has all the upward conversions possible for integer and floating point types.

[2] To place a long into the FPU would require two of these instructions.

Removing Data from the Card

All data is read in 32-bit quantities from a different set of memory-mapped registers that transfers the data (with conversions, if appropriate) from the FPU to the destination specified in the 68xxx instruction. When data is read from a specific location, a hardware state machine on the card retrieves the correct National instruction from PROMs that contain the set of instructions used in our implementation, and generates all the necessary signals to remove the data from an FPU register and place it on the memory bus. An example will serve to illustrate.

Suppose it was desired to place a float that was in FPU register `f0` into the location pointed to by 68xxx register `A2`. The code sequence

```
move.l  ⟨addr⟩,(a2)    ⟨addr⟩ is a memory-mapped read register
```

would initiate a read from the card, and the card would then retrieve the National FPU instruction to place a float from `f0` onto the memory bus[3]. The card then feeds the FPU this instruction, and waits for the FPU to deliver the data from `f0`. It then places the retrieved data on the bus. The instruction retrieved from the PROM is (in National mnemonics) "`MOVF f0,`⟨*memory*⟩"; this National Format 11 instruction is: Move (without conversion) a 32-bit quantity from `f0` into memory.

Once again, it is very important that the operation of moving the data from the FPU onto the bus complete before a bus fault (`DTACK` timeout) occurs. If the card were unable to deliver a result in the allotted time, a bus error would occur due to the fact that the 68xxx would think that no device was present. There is no possibility of inserting a "wait" instruction here because the FPU is delivering a result that is needed by a 68xxx instruction.

The operations of removing data from the card consist of:

- Moving floats and longs from the FPU,
- Moving longs from the FPU, converting them to floats, and
- Reading the 32-bit status/control register.

Note that there are no conversions out of the FPU from any floating point type to integers. This operation cannot be done because it takes the FPU too long to convert a floating point type to an integer; a bus fault (`DTACK` timeout) would occur while the 68xxx instruction is waiting for a result from the card.

There are no conversions out of the FPU from floats into longs because this would involve two 32-bit reads from the same memory-mapped location, which would violate the non-interruptability of the operation. Therefore, a float must be converted to a long in the FPU, and then moved out with two 32-bit moves.

[3] To read a long from the FPU would require two of these instructions.

Performing Operations on the Card

The FPU is instructed to perform operations in which both the source and destination operands are on the chip by executing a "TST.W" instruction to particular memory-mapped locations. All operations initiated via this "TST.W" have FPU registers as their source and destination operands. This keeps the moving of data separate from the operations performed on it, and avoids the problem of interrupts occurring while the FPU is in a funny state expecting data.

Because these operations will not complete before a bus fault occurs, the card relinquishes the bus after the "TST.W" instruction is initiated, and the FPU continues processing. Before another operation can be done to the card, it must complete the current operation; therefore, it is necessary to follow these instructions with some form of a "wait" virtual instruction (see the next section on *Waiting* for more details).

Once the "TST.W" instruction is executed, the card retrieves the National instruction from the PROM containing the set of instructions used in our implementation, and feeds the instruction, along with the necessary signals, to the FPU. An example will serve to illustrate.

Suppose it was desired to add two longs that were in registers `f0` and `f2`. The following sequence accomplishes this.

TST.W ⟨*addr*⟩	(⟨*addr*⟩ is a memory-mapped operation register)
(*wait*)	(wait for the FPU to be done)

Operations initiated with a "TST.W" instruction consist of:

- Moving floats and longs among the FPU registers,
- Performing ADD, SBT, MPY, DVD, NEG, and ABS on floats and longs, and
- Converting between floats and longs among the FPU registers.

Waiting

Some of the operations performed by the FPU can take up to 11 μsec (microseconds) to complete (e.g., the floating point divide). If an operation to the card was initiated while the FPU was busy with a previous operation, data corruption would happen. Therefore, there has to be some mechanism for synchronizing the FPU and the 68xxx. The most obvious method is to have a busy bit on the card and have software loop until the busy bit is cleared. There is however, a better solution, which we call "bogus reads."

The "wait" with the bogus reads is accomplished in the following manner. After an operation is initiated on the card, a number of word-wide reads are done from a user-selected address via the "MOVEM" instruction (see the *Memory Map* section for details on the bogus read addresses). For each of the word reads, the card checks if it is done with the operation in progress. If it is not done, it waits until *just before* a bus fault would happen, and then returns the bus; if it is done, it returns the bus immediately. The number of reads is selectable depending upon the length of time required for the operation—less reads are necessary for a floating point add than a floating point divide. This scheme has two nice features:

- It is a one-instruction wait loop, and
- It is processor-speed independent.

There is another added benefit to this scheme. At the end of the bogus read memory on the card, there is an error bit; it tells whether or not there was an error in the last operation performed by the FPU. By doing one additional word read in the "MOVEM" instruction, this bit may be obtained after the FPU finishes the operation. Hence, in *one* instruction it is possible to both wait for the FPU to finish an operation and obtain information determining whether or not an error has occurred.

The following example will illustrate this. Suppose it was desired to divide the two longs in f0 and f2, and check for a possible error. (In the Pascal Workstation, the floating point card resides in internal I/O space at address $5C0000.)

```
   tst.w    $5C4068          (f0,f1) <- (f2,f3)/(f0,f1)
   movem.w  $5C0018,d0-d4    4 bogus word reads (d0..d3) and
*                            the last read gets the status word
*                            at $5C0020
   btst     #3,d4            the error bit
   bne      error
```

Four bogus word reads are done here assuming a bus fault would occur after 3 μsec and a floating point divide takes 11 μsec. If the card finishes the operation before all of the bogus reads are done, the extra reads are then done at memory speed.

Interrupts

The problem of interrupts and context switches seems to have been solved with the separation of data movement with data operation. On an interrupt, the interrupt handler must merely save the FPU floating point registers, the FPU status/control register, and the card's status byte; if the interrupt is a context switch, all eight FPU floating point registers must be saved. However, what happens if an operation is initiated, and an interrupt occurs before the bogus read can start? Because there is no busy/idle bit on the card, the ISR has no idea if the FPU is active or not!

The obvious solution in the ISR would be to assume the worst; i.e., before doing any operation with the card, wait (using the bogus read mechanism) for the current operation to finish. The longest operation is the long divide, so four bogus reads would need to be done for a maximum wait time of 11 μsec. However, these reads only have to be done if the time from the interrupt to the time of the first floating point operation in the ISR is less than 11 μsec.

Practically, very few ISR's do floating point and those that do presumably are not time critical (or else they would not be doing floating point!). Therefore, the above solution should suffice for practically all applications.

Of course, the FPU registers that are used in the ISR must be saved and restored. See the section *Saving and Restoring Context* on the proper sequence for saving and restoring floating point context.

Memory Map

The 98635A card physically resides at address $5C0000 in Series 200 address space. The following are offsets from the starting address of the card:

$1	This byte is the card ID byte. Reading a byte from this location will return the card's ID, which is 10. Writing a 1 byte to this location will reset the card, which must be done at powerup. Writing a 0 byte to this location will place the card in an idle loop mode; it can be removed from this state only by writing a 1 to this location.
$10-$1F	These locations may be read for the bogus reads.
$21	This is the card status register byte. Bit 3 contains the status of the last operation done by the card; it is cleared if the last operation did not cause an error and is set if an error occurred. Writing a 0 to this location will clear a hardware state machine on the card; this is necessary if doing multiple moves from the card FPU registers.

The next offsets are operation offsets, which will be referred to as *pseudo instructions.* They start at offset $4000. The identifier associated with the offset is a mnemonic patterned after the National instruction format. The first three letters specify the operation to be done:

ADD	Addition
SUB	Subtraction
MUL	Multiplication
DIV	Division
NEG	Negate
ABS	Absolute value
MOV	Move

The next letter(s) specify the size of the operands:

I	Integer
F	Float
L	Long

An underscore terminates this field in the pseudo-instruction. The next two fields (separated by underscores) are the source and destination for the operation. They are referred to as GEN1 and GEN2 in National lingo. The literal "m" specifies memory, and can be either a 68xxx register (if applicable) or a 68xxx memory location. All pseudo-instructions with the literal "m" either write to the card or read from the card, depending upon whether "m" is used as the source or destination operand.

Examples: (all assume that a0 points to the start of the card)

```
tst.w   divl_f0_f6(a0)        (f6,f7) <- (f6,f7) / (f0,f1)

tst.w   divf_f4_f3(a0)        f3 <- f3 / f4

move.l  (sp)+,movif_m_f7(a0)  f7 <- convert integer to float

move.l  movf_f0_m(a0),d5      move the float in f0 into d5
```

For a list of all the possible pseudo-ops, see the list at the end of this chapter.

Creating Pseudo-Instructions

Following are several sets of rules with which you can put together pseudo-instructions. These rules are given in addition to enumerating an exhaustive list of every possible combination of data type, register-register combination and operation as is done at the end of this chapter.

These rules are the rules whereby the list of all supported pseudo-ops in the tables at the end of this section were created. The rules are probably the most convenient way to go from what you want to do to the card offset. The tables are probably the most convenient way to go the other direction (see the section *Debugging* later in this chapter).

NOTE

Except where explicitly noted, all operations require bogus reads.

Long Reals (Longs)

For long real math operations, both the source and the destination registers named must be even numbers, because a long real requires 64 bits (two registers' worth) to be represented.

ADDL	\$4000+4×source+destination
SUBL	\$4020+4×source+destination
MULL	\$4040+4×source+destination
DIVL	\$4060+4×source+destination
NEGL	\$4080+4×source+destination
ABSL	\$40A0+4×source+destination

Example: Suppose you want an instruction which would do a long real multiply of the real number in `f2` and `f3` by the number in `f4` and `f5`. According to the above list, long real multiplication starts with an offset of `$4040`. Since the source register is `f2`, we add 4×2 to the `$4040`. Then, since the destination register is `f4`, we add another 4. The sum is `$404C`, which is the desired offset, and the value of the pseudo-instruction `MULL_f2_f4`.

Short Reals (Floats)

For short real math operations, both the source and the destination registers named may be even or odd numbers, since short reals require only 32 bits to be represented.

`ADDF`	`$40C0`+16×source+2×destination
`SUBF`	`$4140`+16×source+2×destination
`MULF`	`$41C0`+16×source+2×destination
`DIVF`	`$4240`+16×source+2×destination
`NEGF`	`$42C0`+16×source+2×destination
`ABSF`	`$4340`+16×source+2×destination

Example: Suppose you want an instruction which would do a short real subtraction of the real number in `f2` from the number in `f7`. According to the above list, short real subtraction starts with an offset of `$4140`. Since the source register is `f7`, we add 16×7 to the `$4140`. Then, since the destination register is `f2`, we add another 2×2. The sum is `$41B4`, which is the desired offset, and the value of the pseudo-instruction `SUBF_f7_f2`.

Register-Register Moving

In the area of moving data from register to register, one must deal with different combinations of data types implicit to the move; that is, a type conversion can take place after the datum is read and before it is sent to the destination. Thus, there are several sub-headings below, from which you must determine what operation you need, and select the appropriate rule with which to create the desired pseudo-instruction.

Moving Short Reals Into Long Reals

For this operation, that of converting a short real number in one register to a long real number in a pair of registers, the source register number may be either even or odd (shorts require only 32 bits), but the destination register number must be even (longs require 64 bits).

`MOVFL_`⟨*FPU reg*⟩`_`⟨*FPU reg*⟩   `$43C0`+8×source+destination

Example: Suppose you want an instruction which would convert a short real in register `f0` to a long real in registers `f6` and `f7`. According to the above formula, a "move float to long" starts with an offset of `$43C0`. Since the source register is `f0`, we add 8×0 to the `$43C0`. Then, since the destination register is `f6`, we add another 6. The sum is `$43C6`, which is the desired offset, and the value of the pseudo-instruction `MOVFL_f0_f6`.

Moving Long Reals Into Short Reals

For this operation, that of converting a long real number in a pair of registers to a short real number another register, the source register number must be even (longs require 64 bits), but the destination register number may be either even or odd (shorts require only 32 bits).

`MOVLF_`⟨*FPU reg*⟩`_`⟨*FPU reg*⟩   `$43C0`+8×source+2×destination

Example: Suppose you want an instruction which would convert a long real in registers `f4` and `f5` to a short real in register `f7`. According to the above formula, a "move long to float" starts with an offset of `$4400`. Since the source register is `f4`, we add 8×4 to the `$4400`. Then, since the destination register is `f7`, we add another 2×7. The sum is `$442E`, which is the desired offset, and the value of the pseudo-instruction `MOVLF_f4_f7`.

Moving Long Reals Into Long Reals

For this operation, that of moving a long real number in one pair of registers—without conversion—into another pair of registers, both the source register number and the destination register number must be even (longs require 64 bits).

`MOVL_`⟨*FPU reg*⟩`_`⟨*FPU reg*⟩   `$4440`+4×source+destination

Example: Suppose you want an instruction which would move the long real in registers `f2` and `f3` to registers `f6` and `f7`. According to the above formula, a "move long" starts with an offset of `$4440`. Since the source register is `f2`, we add 4×2 to the `$4440`. Then, since the destination register is `f6`, we add another 6. The sum is `$444E`, which is the desired offset, and the value of the pseudo-instruction `MOVL_f2_f6`.

Moving Short Reals Into Short Reals

For this operation, that of moving a short real number in one register to a short real number in another register, both the source register number and the destination register number may be either even or odd (shorts require only 32 bits).

`MOVF_`⟨*FPU reg*⟩`_`⟨*FPU reg*⟩   `$4460`+16×source+2×destination

Example: Suppose you want an instruction which would move the short real in register `f5` to register `f1`. According to the above formula, a "move float" starts with an offset of `$4460`. Since the source register is `f5`, we add 16×5 to the `$4460`. Then, since the destination register is `f1`, we add another 2×1. The sum is `$44B2`, which is the desired offset, and the value of the pseudo-instruction `MOVF_f5_f1`.

Register-Memory Moving

The CPU for the Series 200 machines is the Motorola 68xxx chip. In this section, the phrase "68xxx register" will refer to either a data register in the CPU itself, or the memory area pointed to by an address register in the CPU (any 68xxx addressing mode may be used).

In the area of moving data from FPU register to 68xxx and 68xxx to FPU register, one must deal with different combinations of data types implicit to the move; that is, a type conversion can take place after the datum is read and before it is sent to the destination. So again, there are several sub-headings below, from which you must determine what operation you need, and select the appropriate rule with which to create the desired pseudo-instruction.

Note that by convention, longs held in data registers in the 68xxx typically have their most significant portion in the *even* numbered register (due to the fact that the most significant portion of a long in memory occupies the lower memory address), while in the FPU, the most significant portion of a long is in the *odd* numbered register. For example, `D0` (the most significant portion) is placed into `f1` and `D1` (the least significant portion) is placed into `f0`. This is why in the formulae that follow, amounts are *subtracted* from the base offsets for each operation.

Moving 68xxx Floats to FPU Floats

For this operation, that of moving a short real number in a data register on the 68xxx to a short real number in a register on the FPU, the destination register number may be either even or odd (shorts require only 32 bits).

Note: This operation *does not* require bogus reads.

`MOVF_m_`⟨*FPU reg*⟩	`$44FC`−4×destination

Example: Suppose you want an instruction which would move the short real in the 68xxx to register `f1` on the FPU. According to the above formula, a "move 68xxx float to FPU float" starts with an offset of `$44FC`. Since the destination register is `f1`, we subtract 4×1. The result is `$44F8` which is the desired offset, and the value of the pseudo-instruction `MOVF_m_f1`.

Moving 68xxx Integers to FPU Floats

For this operation, that of moving an integer in a data register on the 68xxx to a short real number in a register on the FPU, the destination register number may be either even or odd (shorts require only 32 bits).

`MOVIF_m_`⟨*FPU reg*⟩	`$451C`−4×destination

Example: Suppose you want an instruction which would move the short real in the 68xxx to register `f4` on the FPU. According to the above formula, a "move 68xxx integer to FPU float" starts with an offset of `$451C`. Since the destination register is `f4`, we subtract 4×4. The result is `$450C` which is the desired offset, and the value of the pseudo-instruction `MOVIF_m_f1`.

Moving 68xxx Integers to FPU Longs

For this operation, that of moving an integer in a data register on the 68xxx to a long real number in a register on the FPU, the destination register number must be even (longs require 64 bits).

`MOVIL_m_`⟨*FPU reg*⟩	`$452C`−2×destination

Example: Suppose you want an instruction which would move an integer in the 68xxx to—after conversion to a long real— registers `f6` and `f6` on the FPU. According to the above formula, a "move 68xxx integer to FPU long" starts with an offset of `$452C`. Since the destination register is `f6`, we subtract 2×6. The result is `$4520` which is the desired offset, and the value of the pseudo-instruction `MOVIL_m_f6`.

Moving 68xxx Floats to FPU Longs

For this operation, that of moving an short real in a data register on the 68xxx to a long real number in a pair of registers on the FPU, the destination register number must be even (longs require 64 bits).

Note: This operation *does not* require bogus reads.

`MOVFL_m_`⟨*FPU reg*⟩ $4453C-2\times$destination

Example: Suppose you want an instruction which would move the short real in the 68xxx to registers `f0` and `f1` on the FPU. According to the above formula, a "move 68xxx float to FPU long" starts with an offset of `$453C`. Since the destination register is `f0`, we subtract 2×0. The result is `$453C` which is the desired offset, and the value of the pseudo-instruction `MOVFL_m_f0`.

Moving FPU Floats to 68xxx Floats

For this operation, that of moving a short real number in a register on the FPU to a data register on the 68xxx, the source register number may be either even or odd (shorts require only 32 bits).

Note: This operation *does not* require bogus reads.

`MOVF_`⟨*FPU reg*⟩`_m` `$456C`$-4\times$source

Example: Suppose you want an instruction which would move the short real in register `f5` on the FPU to the 68xxx. According to the above formula, a "move FPU float to 68xxx float" starts with an offset of `$456C`. Since the source register is `f5`, we subtract 4×5. The result is `$4558` which is the desired offset, and the value of the pseudo-instruction `MOVF_f5_m`.

Moving FPU Longs to 68xxx Floats

For this operation, that of moving a long real number in a pair of registers on the FPU to—after conversion to a float—a data register on the 68xxx, the source register number must be even (longs require 64 bits).

Note: This operation *does not* require bogus reads.

`MOVLF_`⟨*FPU reg*⟩`_m` `$457C`$-2\times$source

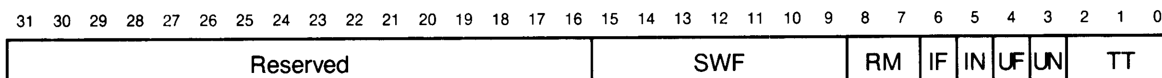
Example: Suppose you want an instruction which would move the long real in registers `f4` and `f5` on the FPU to the 68xxx. According to the above formula, a "move FPU long to 68xxx float" starts with an offset of `$457C`. Since the source register is `f4`, we subtract 2×4. The result is `$4574` which is the desired offset, and the value of the pseudo-instruction `MOVLF_f4_m`.

Status and Control

There are two special pseudo-instructions that involve the FPU's status register (not to be confused with the card's status register). These are pseudo-instructions `LFSR_m_m` and `SFSR_m_m` (neither requires bogus reads).

LFSR_m_m	$4540
SFSR_m_m	$4580

The `SFSR_m_m` pseudo-instruction reads the value of this register. This is typically done after the card status register signals that some form of error has occurred. The `LFSR_m_m` is used to initialize the FPU's 32-bit status register. This status/control register selects operating modes and records any exceptions encountered during the execution of a floating point operation. Its format is given below.



All the fields above are either status fields or control fields. The control fields select FPU operation modes, and the status fields record exceptional conditions during processing. The meanings of these fields follow:

SWF: FSR Software Field (Status)

Bits 15-9 hold and display any information written to them, but are not used by the FPU hardware. They are reserved for use with National's floating point extension software.

The value given the status/control register at Pascal Workstation powerup is $8; this sets the rounding mode to round to nearest even and enables exceptions caused by underflow.

RM: Rounding Mode (Control)

Bit numbers 8 and 7. The allowable rounding modes are:

00	Round to nearest even.
01	Round toward zero.
10	Round toward positive infinity.
11	Round toward negative infinity.

IF: Inexact Result Flag (Status)

Bit 6. This bit is set whenever a result must be rounded. It is set only if no other error has occurred.

IN: Inexact Result Trap Enable (Control)

Bit number 5. If this bit is set, an exception is raised when the result of an operation cannot be represented exactly. If it is not set, the result is rounded according to the rounding mode.

UF: Underflow Flag (Status)

Bit 4. This bit is set whenever an underflow occurs. Its function is not affected by the state of the UN bit.

UN: Underflow Trap Enable (Control)

Bit number 3. If this bit is set, an exception condition will be raised when an underflow occurs. If it is not set, any underflow condition returns a value of 0 while setting the UF flag.

TT: Trap Type (Status)

This 3-bit field indicates cause of an exception. The exception causes are:

0 (000)	No exception
1 (001)	Underflow. This occurs only if the UN bit is set.
2 (010)	Overflow
3 (011)	Divide by zero
4 (100)	Illegal instruction. This occurs if an undefined floating point instruction is passed to the FPU. This exception should never occur—if it does, it implies that the information in the PROMs on the card has been altered and the card is not usable.
5 (101)	Invalid operation. This is caused either by both operands being 0 and executing a divide, or by an IEEE reserved operand being used. This exception can be raised typically by uninitialized variables.
6 (110)	Inexact result. This occurs only if the IN bit is set.

Programming Examples

Following are some examples on how to use the floating-point card from 68xxx assembly language.

Powerup/Reset

The following code initializes the card, and sets the control modes to do rounding to nearest even and signal an exception on underflow.

```
lea      $5C0000,a0            point to the card
move.b   #1,1(a0)              place it in the proper state
move.l   #8,lfsr_m_m(a0)       initialize status/control
```

Error Handling

The following code determines the type of exception that has just occurred. It can be used after an error has been detected by reading the error bit in the card status register. For examples on reading the card's status register, please see the examples under the following section on *Bogus Reads*.

```
lea      $5C0000,a0            point to the card
moveq    #7,d0                 mask for the exception cause
and.l    sfsr_m_m(a0),d0       get the status/control reg
cmp.w    #1,d0                 check for an underflow
beq      err_underflow
cmp.w    #2,d0                 check for an overflow
beq      err_overflow
cmp.w    #3,d0                 check for a divide by zero
beq      err_divbyzero
    .
    .
    .
```

Note that it is not necessary to clear the cause of the exception in the FPU; the next operation performed by the FPU will reset both the FPU status/control register and the card's status register.

Bogus Reads

Bogus reads take place after operations in which both the source and destination operands are the FPU's registers and after operations that convert a 32-bit integer into a floating point operand while placing the floating point result into one of the FPU's registers. The hardware allows up to 10 bogus word reads, but in practice, generally four word reads are done for all operations. Four word reads are enough to ensure that the longest operation (a 64-bit floating point divide) will complete before the last read. Because extraneous bogus reads are done so quickly (at memory speed), there is no performance advantage to be gained by optimizing the reads for every particular operation. In fact, the Pascal Workstation compiler generates code in which four bogus reads are done for every operation that requires bogus reads.

If no check is to be made for a possible exception, a bogus read like the following:

```
movem.l   $5C0018,d6-d7
```

will suffice for all operations that require bogus reads. To check for exceptions, there are two alternatives:

```
movem.l $5C0018,d6-d7   4 bogus word reads
btst    #3,$5C0021      test exception bit in card status
bne     error_handler   branch to decode the error type
```

or

```
movem.w $5C0018,d3-d7   4 bogus word reads plus 1 word read to
btst    #3,d7           get the card status
bne     error_handler
```

In the second example, an additional read is done after the four bogus reads to obtain the status byte in the low byte of D7. The bogus read can also be done with:

```
movem.l $5C0016,d5-d7  5 bogus word reads plus 1 word read
                       to get the card status
btst    #3,d7          get the card status
bne     error_handler
```

In this example, an extra bogus read is done (in addition to the read that obtains the status word) for the purpose of not clobbering so many D-registers. All operations that can produce exceptions require bogus reads.

Moving Data Into the FPU

The 98635A card is designed to accept 32-bit data. It is possible, then, to move floats and longs into the FPU registers, and also to move in integers, converting them into either floats or longs. Bogus reads are required only when converting integers into floating point types. Longs in the FPU registers occupy consecutive register pairs and are addressed with the pseudo-instructions using the even or low-numbered register in the pair. The most significant portion of a long is held in the odd register and the least significant portion is held in the even register.

Suppose `D0` contained a float. To place the contents of `D0` into `f6`, use

```
lea     $5C0000,a0
move.l  d0,movf_m_f6(a0)
```

To move longs into the FPU's registers, either two `MOVE`s or one `MOVEM` is required. Suppose (`D0`,`D1`) contained a long real. To place the contents of (`D0`,`D1`) into the FPU register pair (`f0`,`f1`), use

```
lea     $5C0000,a0
move.l  d0,movf_m_f1(a0)
move.l  d1,movf_m_f0(a0)
```

or

```
lea     $5C0000,a0
movem.l d0-d1,movf_m_f1(a0)
```

Remember that by convention, longs held in D-registers typically have their most significant portion in the *even* numbered register (due to the fact that the most significant portion of a long in memory occupies the lower memory address), while in the FPU, the most significant portion of a long is in the *odd* numbered register. This is why in the above example, `D0` (the most significant portion) is placed into `f1` and `D1` (the least significant portion) is placed into `f0`.

Fortunately, the pseudo-instructions "`MOVF_m_f`x" (x in 0-7) are memory-mapped in consecutive addresses, so it is possible to place up to eight floats or four longs into the FPU registers in one instruction. Suppose (`D0`,`D1`) and (`D2`,`D3`) contain two longs that are to be added together. To place them into FPU register pairs (`f6`,`f7`) and (`f4`,`f5`), use:

```
lea      $5C0000,a0
movem.l d0-d3,movf_m_f7(a0)
```

In this example, the long in (`D0`,`D1`) would be placed into (`f6`,`f7`) and the long in (`D2`,`D3`) would be placed into (`f4`,`f5`). Note that the pseudo-instructions are in increasing offsets from "`MOVF_m_f7`". Therefore, to place all eight D-registers into the FPU, use:

```
lea      $5C0000,a0
movem.l d0-d7,movf_m_f7(a0)
```

The next example will take the integer in `D0`, and place it in `f5`, converting it into a float.

```
lea      $5C0000,a0
move.l   d0,movif_m_f5(a0)
movem.l $18(a0),d6-d7           bogus reads
```

The bogus reads are required in this case because the operation of converting an integer into floating point type will not complete before a bus fault occurs.

The next example will take the integer in `D0`, and place it into (`f2`,`f3`), converting it into a long.

```
lea      $5C0000,a0
move.l   d0,movil_m_f2(a0)
movem.l $18(a0),d6-d7           bogus reads
```

Finally, this example will take the float in `D3`, and place it into (`f6`,`f7`), converting it into a long.

```
lea      $5C0000,a0
move.l   d3,movfl_m_f6(a0)
```

The "`MOVFL`" pseudo-instructions all are memory-mapped in consecutive addresses, so it is possible to do several moves and conversions in one instruction (as in the "`MOVF_m_f`x" pseudo-instructions, where x is in 0-7). Because the "`MOVIF`" and "`MOVIL`" pseudo-instructions require bogus reads, it is not possible to combine several of these operations in one 68xxx instruction.

Moving Data Out of the FPU

Data is moved out of the FPU in 32-bit quantities. To move a float from `f5` into `D6`, use

```
lea     $5C0000,a0
move.l  movf_f5_m(a0),d6
```

To move a long from (`f0`,`f1`) into (`D0`,`D1`), use

```
lea     $5C0000,a0
move.l  movf_f1_m(a0),d0
move.l  movf_f0_m(a0),d1
```

or

```
lea     $5C0000,a0
movem.l movf_f1_m(a0),d0-d1
clr.b   $21(a0)
```

Note how the least significant portion of the long (`f0`) is moved into `D1`, while the most significant portion (`f1`) is moved into `D0`. By convention, longs held in D-registers typically have their most significant portion in the *even* numbered register (due to the fact that the most significant portion of a long in memory occupies the lower memory address), while in the FPU, the most significant portion of a long is in the *odd* numbered register. If longs are removed from the FPU and placed into memory, it is important that the most significant register of the register pair (i.e., the odd register) be placed in the lower memory address.

The "`CLR.B`" instruction is necessary when doing any multiple reads from the card because the "`MOVEM`" instruction causes an additional read to take place, thus triggering an operation on the card. The "`CLR.B`" will abort this operation. Reading the FPU registers with a "`MOVEM`" instruction should *only* be done in a single process environment or when all interrupts are locked out. If an interrupt should occur directly after the "`MOVEM`" instruction, the card will be in a non-savable state. See the section *Saving/Restoring Context* for an example using the "`MOVEM`" instruction to save all eight FPU data registers.

It is also possible to convert longs into floats while removing them from the FPU. In the following example,

```
lea     $5C0000,a0
move.l  movlf_f2_m(a0),d0
```

the long in (`f2`,`f3`) would be converted to a float and placed into `D0`.

Unfortunately, it is not possible to use the card to convert floats or longs into integers. It is possible to convert floats into longs on the FPU, but the result destination must be one of the long register pairs. All conversions from floating point types to integers must be done in software.

Saving and Restoring Context

The following example assumes that the FPU is in a quiescent state. To save the entire context of the floating point card:

```
lea      $5C0000,a0
move.b   $21(a0),-(sp)        save card status byte
move.l   sfsr_m_m(a0),-(sp)   save FPU status/control
move.l   movf_f7_m(a0),-(sp)  save each fp register
move.l   movf_f6_m(a0),-(sp)
       .
       .
       .
move.l movf_f0_m(a0),-(sp)
```

Another way of saving the context (assumes interrupts are locked out):

```
lea      $5C0000,a0
move.b   $21(a0),-(sp)        save status of last operation
move.l   sfsr_m_m(a0),-(sp)   save the FPU status/control
movem.l movf_f7_m(a0),d0-d7  remove the data registers
clr.b    $21(a0)              reset the state machine
movem.l d0-d7,-(sp)          save the 8 data registers
```

This takes advantage of the fact that the pseudo-instructions to move the FPU registers without conversion are memory-mapped in consecutive addresses.

To restore the entire context of the floating point card:

```
lea      $5C0000,a0
movem.l (sp)+,d0-d7            get the 8 data registers
movem.l d0-d7,movf_m_f7(a0)    place back into the FPU
movem.l (sp)+,lfsr_m_m(a0)     restore FPU status/control
move.b   (sp)+,d0               get last operation status
lsr.b    #3,d0                  move error bit for hardware
move.b   d0,$21(a0)             restore status
```

It is **extremely important** that the card status byte at offset `$21` be saved first, before any actual operations to the FPU are initiated; any pseudo-instructions to the FPU will *overwrite* the contents of the card status byte, and thus destroy the error bit. If is also **extremely important** that the FPU status/control register be saved next because the instructions to remove the FPU registers will alter the status/control register (as well as alter the card status byte). Failure to follow this convention can cause incorrect error processing! For example, assume process *A* is executing

```
lea      $5C0000,a0
tst.w    divl_f2_f0(a0)       do a long divide
movem.l bogus4(a0),d4-d5     wait for the chip to finish
btst     #3,$21(a0)           check for an error
jne      error_handler
```

and an interrupt occurs right after the long divide is initiated. If the ISR does not save/restore the card status byte and FPU status/control register correctly, the "BTST" will be testing the result of the last operation in the ISR and not the result of the long divide.

On-Board FPU Operations

All number-crunching is done on the FPU itself, and requires both source and destination to be FPU registers. Operations are initiated via the "TST.W" instruction. The following

```
lea      $5C0000,a0
tst.w    divl_f0_f2(a0)
movem.l  $18(a0),d6-d7          bogus reads
```

will divide the long in (f2,f3) by the long in (f0,f1), and place the long result in (f2,f3). The following

```
lea      $5C0000,a0
tst.w    movlf_f0_f7(a0)
movem.l  $18(a0),d6-d7              bogus reads
```

will take the long in (f0,f1) convert it to a float, and place the result in f7.

Putting it All Together

Following is an example of how to put all of the above information into a code sequence that uses the card.

The following code sequence will add the two longs in (d0,d1) and (d2,d3), and place the result long into (d0,d1):

```
lea      $5C0000,a0             point to the start of the card
movem.l  d0-d3,movf_m_f3(a0)    f2 <- (d0,d1); f0 <- (d2,d3)
tst.w    addl_f0_f2(a0)         (f2,f3) <- (f2,f3) + (f0,f1)
movem.l  $18(a0),d6-d7          4 bogus word reads
btst     #3,$21(a0)             test exception bit in status
bne      error_handler          decode the error type
move.l   movf_f3_m(a0),d0       else place result in (d0,d1)
move.l   move_f2_m(a0),d1
```

Operating System Modifications

All math software that detected a real value of zero had to be modified to also detect a real value of minus zero because the FPU, in conformance with the proposed IEEE standard, produces minus zero for certain operations.

The Pascal Workstation compiler has been modified to generate code to talk directly to the floating point card. Please refer to the appropriate documentation for information on how to use the new compiler.

In the Pascal Workstation, the `ALLREALS` library was modified in two major places. The `ASM_RADD`, `ASM_RSUB`, `ASM_RMUL`, and the `ASM_RDIV` compiler interface routines were modified to detect and use the card. Also, the transcendentals (along with the transcendental polynomial evaluator) were rewritten to make optimum use of the FPU's registers.

If an application program is merely recompiled, the card will be used via the libraries and a performance increase will be seen. However, for maximum performance gain the new compiler directive (`$FLOAT_HDW$`) must be used.

The card can be disabled/enabled using the `$SYSPROG$` directive and the system variable "`FLTPTHDW`" (which is set at powerup) in the following manner:

```
var
  math_type['fltpthdw']:       boolean;
begin
math_type:=false;              {disable the floating point card}
math_type:=true;               {enable the floating point card}
```

An additional error number, −29, was added to the list of system error numbers. It is generated for miscellaneous floating point errors caused by such things as uninitialized variables and operations such as 0÷0.

The compiler saves FPU registers in use upon function or subroutine calls, and restores them upon return.

Differences

One of the major goals of the floating-point card project was to ensure that the presence or absence of the 98635A card would have no direct bearing on the results generated by a program. This goal was met for programs that run without errors; in fact, all test programs indicate that results are identical *bit for bit*[4]. There are, unfortunately, some differences in programs in which errors are generated. If an attempt is made to divide zero by itself, the FPU will signal an "invalid operation", while the floating point software signals a "divide by zero." This error difference will only be seen in Workstation Pascal. Programs run using the floating point card and having uninitialized variables could produce errors, while the same programs running under the software floating point could produce apparently good results. The reason for this is that the FPU will detect invalid IEEE operands, while the floating point software will not.

[4] This is not *entirely* true, since −0 and +0, although "equal," differ in the sign bit!

Debugging

Low-level debugging (e.g., examining the contents of the floating point registers) is possible, although not easy. In debugging the floating point hardware, there are primarily two things that are required. The first thing is recognizing an assembly language instruction that references the hardware, and the second thing is knowing what specific operation is taking place. Once a floating point operation has been recognized, two possible actions can occur: the contents of a floating point register can be examined, or the contents of a floating point register can be altered.

Floating Point Instruction Recognition

Obviously, recognition is no problem in debugging hand written assembly language; it becomes more difficult in examining compiler output. The easiest way to recognize a floating point operation is via the address that the card occupies. The card resides at address `$5C0000`, so any instruction that references addresses greater than this and less that `$5C4582` is using the card. The Pascal Workstation compiler uses absolute addressing in talking to the card; therefore, any instruction that contains an address reference in the range `$5C0000` through `$5C4580` is using the card. For example, compiler output to move a long real from the stack into floating point registers (`f0`,`f1`) might look like

```
move.l     (sp)+,$5C44F8
move.l     (sp)+,$5C44FC
```

The key to recognizing a floating point operation is the fact that the Pascal Workstation compiler always generates references to the floating point card using the absolute addressing mode.

Floating Point Instruction Knowledge

Once an instruction has been recognized as using the card, the next step is determining what the actual operation was. This is where the list of pseudo-instructions is necessary. The following instruction may be encountered:

```
tst.w   $5C429E
```

Examining the pseudo-instruction table at the end of this section, you will see that the offset `$429E` will cause the float in `f7` to be divided by the float in `f5`, and the result will be placed in `f7`.

The compiler also issues bogus read and error checking instructions. Refer to the memory map section for the offsets to determine what is being done.

Reading and Altering The Floating Point Registers

To examine the registers, it is necessary to "issue" a pseudo-instruction that reads the FPU registers and places them into memory. Because the card is memory-mapped, this is done by reading from the proper card addresses.

There are two ways to examine a floating point register; with the "D" debugger command or the "OL" debugger command. The "D" command must be used with the "R" format, or data *will* be altered on the card! For example, the following instruction

```
tst.w    $5C4002
```

causes the two longs in (f0,f1) and (f2,f3) to be added together, with the result in (f2,f3). To examine the result in (f2,f3), type:

```
D $5C4560^:R
```

while in the debugger. This will display the long in decimal real format. The values must be read with the odd register pseudo-instructions because the odd FPU registers contain the most significant part of the longs, and the system software that converts longs into decimal strings expects to get the most significant part of a long first.

To examine the value in IEEE format, two OLs must be issued:

```
OL $5C4560^
OL $5C4564^
```

The first OL gets the most significant 32 bits of the long, while the second OL gets the least significant 32 bits.

The OL and D with the R format are the only ways to read the floating point registers. If any other commands/formats are used, data on the card will be corrupted.

To alter the registers, use the OL command with the pseudo-instruction offsets "MOVF_m_f*x*" (*x* in 0-7); to alter the FPU status register, use the OL command with the "SFSR_m_m" pseudo-instruction offset. For example, to change the value of FPU register f0, do a

```
OL $5C44FC^  ⟨newvalue⟩
```

The pseudo-instruction at offset $44fC is "MOVF_m_f0".

Pseudo-Instruction Table

Following are all the defined pseudo-instructions defined in the `ALLREALS` module, and thus, the following offsets from the beginning of the floating-point card are the only ones for which the PROM will supply a valid command to the FPU.

Long-Real Operations

addl_fx_fx

addl_f0_f0	$4000	addl_f4_f0	$4010
addl_f0_f2	$4002	addl_f4_f2	$4012
addl_f0_f4	$4004	addl_f4_f4	$4014
addl_f0_f6	$4006	addl_f4_f6	$4016
addl_f2_f0	$4008	addl_f6_f0	$4018
addl_f2_f2	$400A	addl_f6_f2	$401A
addl_f2_f4	$400C	addl_f6_f4	$401C
addl_f2_f6	$400E	addl_f6_f6	$401E

subl_fx_fx

subl_f0_f0	$4020	subl_f4_f0	$4030
subl_f0_f2	$4022	subl_f4_f2	$4032
subl_f0_f4	$4024	subl_f4_f4	$4034
subl_f0_f6	$4026	subl_f4_f6	$4036
subl_f2_f0	$4028	subl_f6_f0	$4038
subl_f2_f2	$402A	subl_f6_f2	$403A
subl_f2_f4	$402C	subl_f6_f4	$403C
subl_f2_f6	$402E	subl_f6_f6	$403E

mull_fx_fx

mull_f0_f0	$4040	mull_f4_f0	$4050
mull_f0_f2	$4042	mull_f4_f2	$4052
mull_f0_f4	$4044	mull_f4_f4	$4054
mull_f0_f6	$4046	mull_f4_f6	$4056
mull_f2_f0	$4048	mull_f6_f0	$4058
mull_f2_f2	$404A	mull_f6_f2	$405A
mull_f2_f4	$404C	mull_f6_f4	$405C
mull_f2_f6	$404E	mull_f6_f6	$405E

divl_fx_fx

divl_f0_f0	$4060	divl_f4_f0	$4070
divl_f0_f2	$4062	divl_f4_f2	$4072
divl_f0_f4	$4064	divl_f4_f4	$4074
divl_f0_f6	$4066	divl_f4_f6	$4076
divl_f2_f0	$4068	divl_f6_f0	$4078
divl_f2_f2	$406A	divl_f6_f2	$407A
divl_f2_f4	$406C	divl_f6_f4	$407C
divl_f2_f6	$406E	divl_f6_f6	$407E

negl_fx_fx

negl_f0_f0	$4080
negl_f0_f2	$4082
negl_f0_f4	$4084
negl_f0_f6	$4086
negl_f2_f0	$4088
negl_f2_f2	$408A
negl_f2_f4	$408C
negl_f2_f6	$408E
negl_f4_f0	$4090
negl_f4_f2	$4092
negl_f4_f4	$4094
negl_f4_f6	$4096
negl_f6_f0	$4098
negl_f6_f2	$409A
negl_f6_f4	$409C
negl_f6_f6	$409E

absl_fx_fx

absl_f0_f0	$40A0
absl_f0_f2	$40A2
absl_f0_f4	$40A4
absl_f0_f6	$40A6
absl_f2_f0	$40A8
absl_f2_f2	$40AA
absl_f2_f4	$40AC
absl_f2_f6	$40AE
absl_f4_f0	$40B0
absl_f4_f2	$40B2
absl_f4_f4	$40B4
absl_f4_f6	$40B6
absl_f6_f0	$40B8
absl_f6_f2	$40BA
absl_f6_f4	$40BC
absl_f6_f6	$40BE

Short-Real Operations

addf_fx_fx

addf_f0_f0	$40C0
addf_f0_f1	$40C2
addf_f0_f2	$40C4
addf_f0_f3	$40C6
addf_f0_f4	$40C8
addf_f0_f5	$40CA
addf_f0_f6	$40CC
addf_f0_f7	$40CE
addf_f1_f0	$40D0
addf_f1_f1	$40D2
addf_f1_f2	$40D4
addf_f1_f3	$40D6
addf_f1_f4	$40D8
addf_f1_f5	$40DA
addf_f1_f6	$40DC
addf_f1_f7	$40DE
addf_f2_f0	$40E0
addf_f2_f1	$40E2
addf_f2_f2	$40E4
addf_f2_f3	$40E6
addf_f2_f4	$40E8
addf_f2_f5	$40EA
addf_f2_f6	$40EC
addf_f2_f7	$40EE
addf_f3_f0	$40F0
addf_f3_f1	$40F2
addf_f3_f2	$40F4
addf_f3_f3	$40F6
addf_f3_f4	$40F8
addf_f3_f5	$40FA
addf_f3_f6	$40FC
addf_f3_f7	$40FE
addf_f4_f0	$4100
addf_f4_f1	$4102
addf_f4_f2	$4104
addf_f4_f3	$4106
addf_f4_f4	$4108
addf_f4_f5	$410A
addf_f4_f6	$410C
addf_f4_f7	$410E
addf_f5_f0	$4110
addf_f5_f1	$4112
addf_f5_f2	$4114
addf_f5_f3	$4116
addf_f5_f4	$4118
addf_f5_f5	$411A
addf_f5_f6	$411C
addf_f5_f7	$411E
addf_f6_f0	$4120
addf_f6_f1	$4122
addf_f6_f2	$4124
addf_f6_f3	$4126
addf_f6_f4	$4128
addf_f6_f5	$412A
addf_f6_f6	$412C
addf_f6_f7	$412E
addf_f7_f0	$4130
addf_f7_f1	$4132
addf_f7_f2	$4134
addf_f7_f3	$4136
addf_f7_f4	$4138
addf_f7_f5	$413A
addf_f7_f6	$413C
addf_f7_f7	$413E

subf_f*x*_f*x*

subf_f0_f0	$4140	subf_f4_f0	$4180
subf_f0_f1	$4142	subf_f4_f1	$4182
subf_f0_f2	$4144	subf_f4_f2	$4184
subf_f0_f3	$4146	subf_f4_f3	$4186
subf_f0_f4	$4148	subf_f4_f4	$4188
subf_f0_f5	$414A	subf_f4_f5	$418A
subf_f0_f6	$414C	subf_f4_f6	$418C
subf_f0_f7	$414E	subf_f4_f7	$418E
subf_f1_f0	$4150	subf_f5_f0	$4190
subf_f1_f1	$4152	subf_f5_f1	$4192
subf_f1_f2	$4154	subf_f5_f2	$4194
subf_f1_f3	$4156	subf_f5_f3	$4196
subf_f1_f4	$4158	subf_f5_f4	$4198
subf_f1_f5	$415A	subf_f5_f5	$419A
subf_f1_f6	$415C	subf_f5_f6	$419C
subf_f1_f7	$415E	subf_f5_f7	$419E
subf_f2_f0	$4160	subf_f6_f0	$41A0
subf_f2_f1	$4162	subf_f6_f1	$41A2
subf_f2_f2	$4164	subf_f6_f2	$41A4
subf_f2_f3	$4166	subf_f6_f3	$41A6
subf_f2_f4	$4168	subf_f6_f4	$41A8
subf_f2_f5	$416A	subf_f6_f5	$41AA
subf_f2_f6	$416C	subf_f6_f6	$41AC
subf_f2_f7	$416E	subf_f6_f7	$41AE
subf_f3_f0	$4170	subf_f7_f0	$41B0
subf_f3_f1	$4172	subf_f7_f1	$41B2
subf_f3_f2	$4174	subf_f7_f2	$41B4
subf_f3_f3	$4176	subf_f7_f3	$41B6
subf_f3_f4	$4178	subf_f7_f4	$41B8
subf_f3_f5	$417A	subf_f7_f5	$41BA
subf_f3_f6	$417C	subf_f7_f6	$41BC
subf_f3_f7	$417E	subf_f7_f7	$41BE

mulf_f*x*_f*x*

mulf_f0_f0	$41C0	mulf_f4_f0	$4200
mulf_f0_f1	$41C2	mulf_f4_f1	$4202
mulf_f0_f2	$41C4	mulf_f4_f2	$4204
mulf_f0_f3	$41C6	mulf_f4_f3	$4206
mulf_f0_f4	$41C8	mulf_f4_f4	$4208
mulf_f0_f5	$41CA	mulf_f4_f5	$420A
mulf_f0_f6	$41CC	mulf_f4_f6	$420C
mulf_f0_f7	$41CE	mulf_f4_f7	$420E
mulf_f1_f0	$41D0	mulf_f5_f0	$4210
mulf_f1_f1	$41D2	mulf_f5_f1	$4212
mulf_f1_f2	$41D4	mulf_f5_f2	$4214
mulf_f1_f3	$41D6	mulf_f5_f3	$4216
mulf_f1_f4	$41D8	mulf_f5_f4	$4218
mulf_f1_f5	$41DA	mulf_f5_f5	$421A
mulf_f1_f6	$41DC	mulf_f5_f6	$421C
mulf_f1_f7	$41DE	mulf_f5_f7	$421E
mulf_f2_f0	$41E0	mulf_f6_f0	$4220
mulf_f2_f1	$41E2	mulf_f6_f1	$4222
mulf_f2_f2	$41E4	mulf_f6_f2	$4224
mulf_f2_f3	$41E6	mulf_f6_f3	$4226
mulf_f2_f4	$41E8	mulf_f6_f4	$4228
mulf_f2_f5	$41EA	mulf_f6_f5	$422A
mulf_f2_f6	$41EC	mulf_f6_f6	$422C

mulf_f2_f7	$41EE
mulf_f3_f0	$41F0
mulf_f3_f1	$41F2
mulf_f3_f2	$41F4
mulf_f3_f3	$41F6
mulf_f3_f4	$41F8
mulf_f3_f5	$41FA
mulf_f3_f6	$41FC
mulf_f3_f7	$41FE
mulf_f6_f7	$422E
mulf_f7_f0	$4230
mulf_f7_f1	$4232
mulf_f7_f2	$4234
mulf_f7_f3	$4236
mulf_f7_f4	$4238
mulf_f7_f5	$423A
mulf_f7_f6	$423C
mulf_f7_f7	$423E

divf_fx_fx

divf_f0_f0	$4240
divf_f0_f1	$4242
divf_f0_f2	$4244
divf_f0_f3	$4246
divf_f0_f4	$4248
divf_f0_f5	$424A
divf_f0_f6	$424C
divf_f0_f7	$424E
divf_f1_f0	$4250
divf_f1_f1	$4252
divf_f1_f2	$4254
divf_f1_f3	$4256
divf_f1_f4	$4258
divf_f1_f5	$425A
divf_f1_f6	$425C
divf_f1_f7	$425E
divf_f2_f0	$4260
divf_f2_f1	$4262
divf_f2_f2	$4264
divf_f2_f3	$4266
divf_f2_f4	$4268
divf_f2_f5	$426A
divf_f2_f6	$426C
divf_f2_f7	$426E
divf_f3_f0	$4270
divf_f3_f1	$4272
divf_f3_f2	$4274
divf_f3_f3	$4276
divf_f3_f4	$4278
divf_f3_f5	$427A
divf_f3_f6	$427C
divf_f3_f7	$427E
divf_f4_f0	$4280
divf_f4_f1	$4282
divf_f4_f2	$4284
divf_f4_f3	$4286
divf_f4_f4	$4288
divf_f4_f5	$428A
divf_f4_f6	$428C
divf_f4_f7	$428E
divf_f5_f0	$4290
divf_f5_f1	$4292
divf_f5_f2	$4294
divf_f5_f3	$4296
divf_f5_f4	$4298
divf_f5_f5	$429A
divf_f5_f6	$429C
divf_f5_f7	$429E
divf_f6_f0	$42A0
divf_f6_f1	$42A2
divf_f6_f2	$42A4
divf_f6_f3	$42A6
divf_f6_f4	$42A8
divf_f6_f5	$42AA
divf_f6_f6	$42AC
divf_f6_f7	$42AE
divf_f7_f0	$42B0
divf_f7_f1	$42B2
divf_f7_f2	$42B4
divf_f7_f3	$42B6
divf_f7_f4	$42B8
divf_f7_f5	$42BA
divf_f7_f6	$42BC
divf_f7_f7	$42BE

negf_fx_fx

negf_f0_f0	$42C0
negf_f0_f1	$42C2
negf_f0_f2	$42C4
negf_f0_f3	$42C6
negf_f0_f4	$42C8
negf_f0_f5	$42CA
negf_f0_f6	$42CC
negf_f0_f7	$42CE
negf_f1_f0	$42D0
negf_f1_f1	$42D2
negf_f1_f2	$42D4
negf_f1_f3	$42D6
negf_f1_f4	$42D8
negf_f4_f0	$4300
negf_f4_f1	$4302
negf_f4_f2	$4304
negf_f4_f3	$4306
negf_f4_f4	$4308
negf_f4_f5	$430A
negf_f4_f6	$430C
negf_f4_f7	$430E
negf_f5_f0	$4310
negf_f5_f1	$4312
negf_f5_f2	$4314
negf_f5_f3	$4316
negf_f5_f4	$4318

negf_f1_f5	$42DA
negf_f1_f6	$42DC
negf_f1_f7	$42DE
negf_f2_f0	$42E0
negf_f2_f1	$42E2
negf_f2_f2	$42E4
negf_f2_f3	$42E6
negf_f2_f4	$42E8
negf_f2_f5	$42EA
negf_f2_f6	$42EC
negf_f2_f7	$42EE
negf_f3_f0	$42F0
negf_f3_f1	$42F2
negf_f3_f2	$42F4
negf_f3_f3	$42F6
negf_f3_f4	$42F8
negf_f3_f5	$42FA
negf_f3_f6	$42FC
negf_f3_f7	$42FE
negf_f5_f5	$431A
negf_f5_f6	$431C
negf_f5_f7	$431E
negf_f6_f0	$4320
negf_f6_f1	$4322
negf_f6_f2	$4324
negf_f6_f3	$4326
negf_f6_f4	$4328
negf_f6_f5	$432A
negf_f6_f6	$432C
negf_f6_f7	$432E
negf_f7_f0	$4330
negf_f7_f1	$4332
negf_f7_f2	$4334
negf_f7_f3	$4336
negf_f7_f4	$4338
negf_f7_f5	$433A
negf_f7_f6	$433C
negf_f7_f7	$433E

absf_f*x*_f*x*

absf_f0_f0	$4340
absf_f0_f1	$4342
absf_f0_f2	$4344
absf_f0_f3	$4346
absf_f0_f4	$4348
absf_f0_f5	$434A
absf_f0_f6	$434C
absf_f0_f7	$434E
absf_f1_f0	$4350
absf_f1_f1	$4352
absf_f1_f2	$4354
absf_f1_f3	$4356
absf_f1_f4	$4358
absf_f1_f5	$435A
absf_f1_f6	$435C
absf_f1_f7	$435E
absf_f2_f0	$4360
absf_f2_f1	$4362
absf_f2_f2	$4364
absf_f2_f3	$4366
absf_f2_f4	$4368
absf_f2_f5	$436A
absf_f2_f6	$436C
absf_f2_f7	$436E
absf_f3_f0	$4370
absf_f3_f1	$4372
absf_f3_f2	$4374
absf_f3_f3	$4376
absf_f3_f4	$4378
absf_f3_f5	$437A
absf_f3_f6	$437C
absf_f3_f7	$437E
absf_f4_f0	$4380
absf_f4_f1	$4382
absf_f4_f2	$4384
absf_f4_f3	$4386
absf_f4_f4	$4388
absf_f4_f5	$438A
absf_f4_f6	$438C
absf_f4_f7	$438E
absf_f5_f0	$4390
absf_f5_f1	$4392
absf_f5_f2	$4394
absf_f5_f3	$4396
absf_f5_f4	$4398
absf_f5_f5	$439A
absf_f5_f6	$439C
absf_f5_f7	$439E
absf_f6_f0	$43A0
absf_f6_f1	$43A2
absf_f6_f2	$43A4
absf_f6_f3	$43A6
absf_f6_f4	$43A8
absf_f6_f5	$43AA
absf_f6_f6	$43AC
absf_f6_f7	$43AE
absf_f7_f0	$43B0
absf_f7_f1	$43B2
absf_f7_f2	$43B4
absf_f7_f3	$43B6
absf_f7_f4	$43B8
absf_f7_f5	$43BA
absf_f7_f6	$43BC
absf_f7_f7	$43BE

Conversion-Moves Between FPU Registers

movfl_fx_fx

movfl_f0_f0	$43C0	movfl_f4_f0	$43E0
movfl_f0_f2	$43C2	movfl_f4_f2	$43E2
movfl_f0_f4	$43C4	movfl_f4_f4	$43E4
movfl_f0_f6	$43C6	movfl_f4_f6	$43E6
movfl_f1_f0	$43C8	movfl_f5_f0	$43E8
movfl_f1_f2	$43CA	movfl_f5_f2	$43EA
movfl_f1_f4	$43CC	movfl_f5_f4	$43EC
movfl_f1_f6	$43CE	movfl_f5_f6	$43EE
movfl_f2_f0	$43D0	movfl_f6_f0	$43F0
movfl_f2_f2	$43D2	movfl_f6_f2	$43F2
movfl_f2_f4	$43D4	movfl_f6_f4	$43F4
movfl_f2_f6	$43D6	movfl_f6_f6	$43F6
movfl_f3_f0	$43D8	movfl_f7_f0	$43F8
movfl_f3_f2	$43DA	movfl_f7_f2	$43FA
movfl_f3_f4	$43DC	movfl_f7_f4	$43FC
movfl_f3_f6	$43DE	movfl_f7_f6	$43FE

movlf_fx_fx

movlf_f0_f0	$4400	movlf_f4_f0	$4420
movlf_f0_f1	$4402	movlf_f4_f1	$4422
movlf_f0_f2	$4404	movlf_f4_f2	$4424
movlf_f0_f3	$4406	movlf_f4_f3	$4426
movlf_f0_f4	$4408	movlf_f4_f4	$4428
movlf_f0_f5	$440A	movlf_f4_f5	$442A
movlf_f0_f6	$440C	movlf_f4_f6	$442C
movlf_f0_f7	$440E	movlf_f4_f7	$442E
movlf_f2_f0	$4410	movlf_f6_f0	$4430
movlf_f2_f1	$4412	movlf_f6_f1	$4432
movlf_f2_f2	$4414	movlf_f6_f2	$4434
movlf_f2_f3	$4416	movlf_f6_f3	$4436
movlf_f2_f4	$4418	movlf_f6_f4	$4438
movlf_f2_f5	$441A	movlf_f6_f5	$443A
movlf_f2_f6	$441C	movlf_f6_f6	$443C
movlf_f2_f7	$441E	movlf_f6_f7	$443E

Non-Conversion Moves Between FPU Registers

movl_fx_fx

movl_f0_f0	$4440	movl_f4_f0	$4450
movl_f0_f2	$4442	movl_f4_f2	$4452
movl_f0_f4	$4444	movl_f4_f4	$4454
movl_f0_f6	$4446	movl_f4_f6	$4456
movl_f2_f0	$4448	movl_f6_f0	$4458
movl_f2_f2	$444A	movl_f6_f2	$445A
movl_f2_f4	$444C	movl_f6_f4	$445C
movl_f2_f6	$444E	movl_f6_f6	$445E

movf_fx_fx

movf_f0_f0	$4460
movf_f0_f1	$4462
movf_f0_f2	$4464
movf_f0_f3	$4466
movf_f0_f4	$4468
movf_f0_f5	$446A
movf_f0_f6	$446C
movf_f0_f7	$446E
movf_f1_f0	$4470
movf_f1_f1	$4472
movf_f1_f2	$4474
movf_f1_f3	$4476
movf_f1_f4	$4478
movf_f1_f5	$447A
movf_f1_f6	$447C
movf_f1_f7	$447E
movf_f2_f0	$4480
movf_f2_f1	$4482
movf_f2_f2	$4484
movf_f2_f3	$4486
movf_f2_f4	$4488
movf_f2_f5	$448A
movf_f2_f6	$448C
movf_f2_f7	$448E
movf_f3_f0	$4490
movf_f3_f1	$4492
movf_f3_f2	$4494
movf_f3_f3	$4496
movf_f3_f4	$4498
movf_f3_f5	$449A
movf_f3_f6	$449C
movf_f3_f7	$449E
movf_f4_f0	$44A0
movf_f4_f1	$44A2
movf_f4_f2	$44A4
movf_f4_f3	$44A6
movf_f4_f4	$44A8
movf_f4_f5	$44AA
movf_f4_f6	$44AC
movf_f4_f7	$44AE
movf_f5_f0	$44B0
movf_f5_f1	$44B2
movf_f5_f2	$44B4
movf_f5_f3	$44B6
movf_f5_f4	$44B8
movf_f5_f5	$44BA
movf_f5_f6	$44BC
movf_f5_f7	$44BE
movf_f6_f0	$44C0
movf_f6_f1	$44C2
movf_f6_f2	$44C4
movf_f6_f3	$44C6
movf_f6_f4	$44C8
movf_f6_f5	$44CA
movf_f6_f6	$44CC
movf_f6_f7	$44CE
movf_f7_f0	$44D0
movf_f7_f1	$44D2
movf_f7_f2	$44D4
movf_f7_f3	$44D6
movf_f7_f4	$44D8
movf_f7_f5	$44DA
movf_f7_f6	$44DC
movf_f7_f7	$44DE

Moves Between 68xxx and FPU

movf_m_fx

movf_m_f7	$44E0
movf_m_f6	$44E4
movf_m_f5	$44E8
movf_m_f4	$44EC
movf_m_f3	$44F0
movf_m_f2	$44F4
movf_m_f1	$44F8
movf_m_f0	$44FC

movif_m_fx

movif_m_f7	$4500
movif_m_f6	$4504
movif_m_f5	$4508
movif_m_f4	$450C
movif_m_f3	$4510
movif_m_f2	$4514
movif_m_f1	$4518
movif_m_f0	$451C

movil_m_fx

movil_m_f6	$4520
movil_m_f4	$4524
movil_m_f2	$4528
movil_m_f0	$452C
movfl_m_f6	$4530
movfl_m_f4	$4534
movfl_m_f2	$4538
movfl_m_f0	$453C

movf_fx_m

movf_f7_m	$4550	movf_f3_m	$4560
movf_f6_m	$4554	movf_f2_m	$4564
movf_f5_m	$4558	movf_f1_m	$4568
movf_f4_m	$455C	movf_f0_m	$456C

movlf_fx_m

movlf_f6_m	$4570	movlf_f2_m	$4578
movlf_f4_m	$4574	movlf_f0_m	$457C

Object Code Format 14

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 two things: a directory describing its contents, and module(s) 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 68xxx register `A5` points.
- **(Loader) Symbol:** The name given to a DEFed 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, which are exported from modules SYSDEVS and LOADER in file INTERFACE.

```
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];
  filname = 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..maxint;            {0..maxlongdir}

  direntry =
    record
      dfirstblk: shortint;         {module starting block}
      dlastblk: shortint;          {block following end}
  {NOTE:  for DIR[0], fields below 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: filname;         {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 (`DFIRSTBLK`), 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	
. . .	

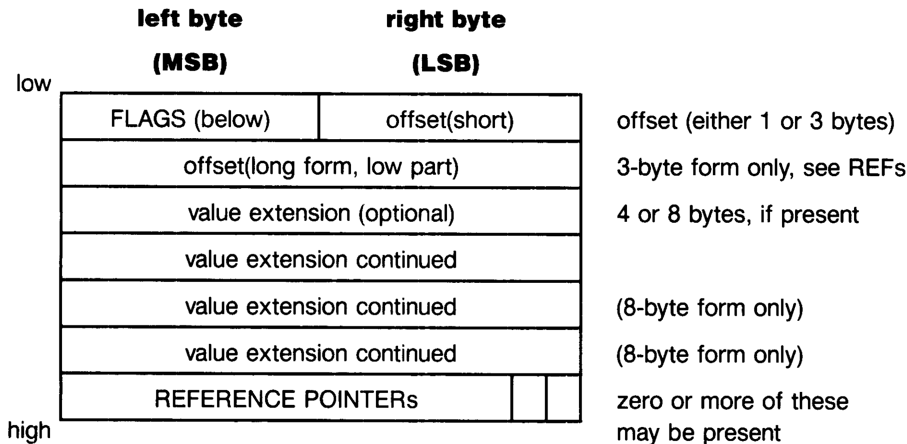
There is a module directory for each module, telling where to find the components of the module; these are different from the directory of the library, which lists the modules themselves. The order in which the parts of a module occur is not specified; the module's directory tells where to find each 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 and therefore will not be found on the Sample Programs disc `EXAMPL:`.

```
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}
  relocatablesize,             {number of relocatable bytes required}
  relocatablebase,             {current origin of relocatable code}
  globalsize,                  {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 (GVRs, 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                 {1 or 3 byte offset}
      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;
```

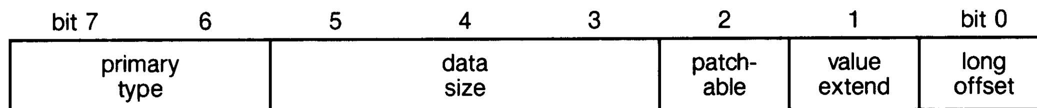
```
gvr = concatenation                            {MOCK PASCAL}
        generalvalue;                   { 2 to 4 bytes of header info}
        valueextension;                    { 0, 4 or 8 bytes of value}
        array [zero or more]                  {list of EXT references}
                of referenceptr;
      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



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

bit 15 thru bit 2	bit 1	bit 0
address of an EXT record relative to beginning of EXT table	add or sub	end flag

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 `0000 0000 0000 00`*xx*) refers to the relocation DELTA for the current module (i.e. new load address minus the old load address).
- **Address 4** (bit pattern `0000 0000 0000 01`*xx*) refers to the global area DELTA for the current module (i.e. new data address minus the old data address).
- **Address 8** (bit pattern `0000 0000 0000 10`*xx*) 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 `0000 0000 0000 0001`, 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 `0000 0000 0000 0101`, 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 GVRs 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 65 532 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	O
L	padding

high

EXT record
This one is 8 bytes long.
The formula is:
EXT size = len +4− (len mod 4)

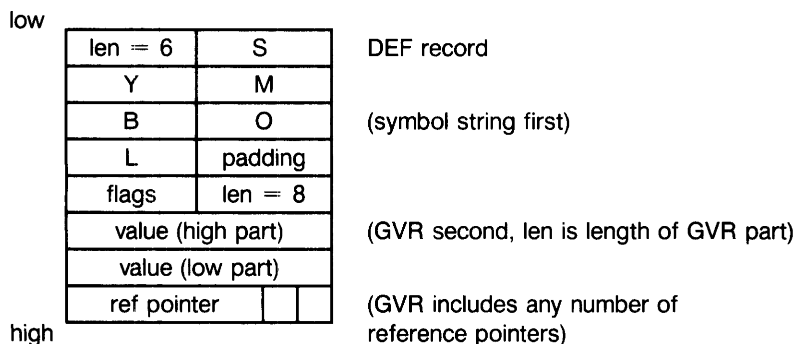
DEF Table (Definition Symbol Table)

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.



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

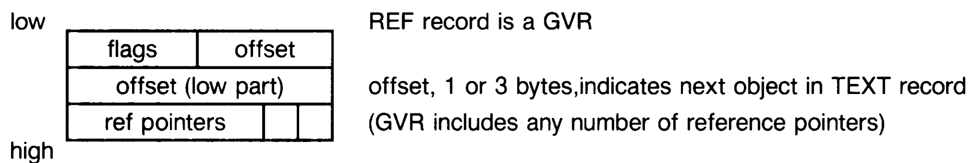
Each REF table follows a TEXT record and is associated with that TEXT record. The REF table begins on a block boundary, which is specified in the directory for the module. Its length is also given in the directory. The REF table is contiguous over its length, which means that individual REF records within 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 DATASIZE 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.



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.

Device I/O 15

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 3.0 Procedure Library* 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:

Address	Region
$FFFFFF $800000	RAM
$7FFFFF $600000	External I/O
$5FFFFF $400000	Internal I/O
$3FFFFF $000000	ROM - system and BOOT ROM (Model 237 frame buffer: typically $300000-$3FFFFF)

The internal interfaces are in the $400000 through $5FFFFF range. These interfaces include the built-in disk drives, keyboard, and CRT devices. This area also includes the built-in HP-IB interface and the plug in DMA interfaces (HP98620A/B). Although the actual locations vary, the intent is that these interfaces will occur on 65 536-byte ($FFFF) boundaries.

The external interfaces (e.g., HP98624A, et cetera) exist in the $600000 through $7FFFFF address range. These interfaces also occur on 65 536-byte ($FFFF) boundaries. As of this writing, only the HP98627 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 (HP98624A). Handling the internal HP-IB interface any other way would cause unnecessary overhead.

The Model 216/217 computers have built-in RS-232 interfaces. This interface is at address $690000 and so is an "external" interface and requires no special handling. This interface is almost 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. The Pascal system 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 this scheme there is only 31 logical select codes presented to the user; however, 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. 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	-	-

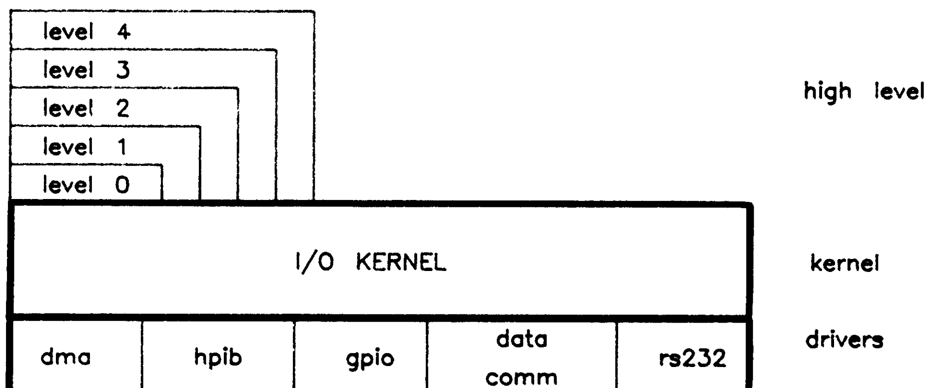
In addition to the these modules there are two other modules of interest to the user: IODECLARATIONS and IOCOMASM. IODECLARATIONS contains the library's constant, type, and variable declarations. IOCOMASM is a support module that contains binary operations (like BINAND, etc.).

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. I/O 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 usually only be two primary places for you to put these modules—INITLIB on the BOOT: volume and the system library. The normal user documentation explains how to configure your system appropriately.

The kernel code and the installed driver code should reside in INITLIB. The kernel export text, I/O library code, and I/O library export text (originally in file IO on the LIB: volume) should be installed in the system library. 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 system library. 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 and makes a call to MARKUSER. 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 calls a procedure in the module which 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 (HP-IB drivers will search for the 98624 interface). This procedure will then set up the driver tables to point to the correct drivers. The program does a MARKUSER.

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 (HP-IB and GPIO in particular).

The IODECLARATIONS module and driver modules exist as object code without EXPORT text. Their export text (if they have export text) is in file INTERFACE. 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	3752	1056	INITLIB	(N)	KERNEL	(P)
				(N)	COMASM	(A)
HPIB	4166	120	INITLIB	(N)	HPIB	(A)
				(N)	H_DRV	(P)
GPIO	2340	120	INITLIB *	(N)	GPIO	(A)
				(N)	G_DRV	(P)
DATA_COMM	5564	126	INITLIB *	(N)	DC	(A)
				(N)	DC_DRV	(P)
DMA	546	0	INITLIB	(N)	DMA_DRV	(P)
RS232	3362	120	INITLIB *	(N)	RS_DRV	(P)
				(N)	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	2046	0	LIBRARY **	(Y)	IOLIB	(P)
HPIB_2	1074	0	LIBRARY **	(Y)	IOLIB	(P)
HPIB_3	604	0	LIBRARY **	(Y)	IOLIB	(P)
SERIAL_0	1646	0	LIBRARY **	(Y)	IOLIB	(P)
SERIAL_3	2974	0	LIBRARY **	(Y)	IOLIB	(P)

* This module might be in INITLIB or on the CONFIG disk.

** This module might be in the system library or in IO on the LIB disk.

The modules IODECLARATIONS, IOCOMASM and KERNEL_INITIALIZE exist in IO as codeless entities for the express purpose of providing user-accessible export text.

Main Data Structures

The way 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 three 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; however, 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 the low-level drivers are.
2. Where the driver R/W for this select code is.
3. What sort of interface this is.
4. Where the interface card is.

The table to support these needs is defined as:

```
CONST
  iominisc   = 0 ;                 { 0 - 7  internal }
  iomaxisc   = 31;                 { 8 - 31 external 8..31 }
TYPE
  type_isc        = iominisc..iomaxisc ;
  isc_table_type  = RECORD
                      io_drv_ptr: ^drv_table_type;   (where are the drivers?)
                      io_tmp_ptr: pio_tmp_ptr;       (where is the driver 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;
VAR
  isc_table       : PACKED ARRAY [type_isc]
                    OF isc_table_type;
```

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 HP-IB 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 ;
bubble_card    = 8 ;
eprom_prgmr    = 9 ;
```

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 HP98626). 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_dsndl   = -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 }
hpgator         = 25;           { bit-mapped display }
hp98253         = 27;           { EPROM programmer   }
hp98627         = 28;           { graphics        }
hp98259         = 30;           { bubble memory }
hp98644         = 66;           {serial:
                                   pri. id: 2;
                                   sec. id: 2    }
```

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_IDs.

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 two 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. For most applications, the 24 external available select codes should be 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 ;      { request 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_1
serial_DMA
serial_FHS
serial_FASTEST
dummy_tfr_2
OVERLAP
overlap_FASTEST
overlap_FHS
overlap_DMA
overlap_INTR
```

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 to 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 parameter 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 Model 226 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, and 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 HP-IB 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 HP-IB-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 lookup there—rather than take the time to pass it in as a parameter.

Notice that the system ISRIB is the first field in the driver read/write (IO_TEMP_PTR^). This is done because when an interrupt strikes, the ISR gets called with only one argument: the address of the ISRIB that contains the ISR procedure variable. Making the ISRIB be the first thing in driver read/write is particularly convenient, because the address of the ISRIB is also the address of driver read/write area for the card. This allows the ISR easy access to the information in the read/write area.

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 straightforward. 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 (e.g., 7) or a device specifier (e.g., 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:

```
addr_to_listen        set_to_listen
addr_to_talk          set_to_talk
```

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_ routines are intended for use by bus control routines (like TRIGGER and others in HPIB_2 and HPIB_3). The ADDR_TO_ routines 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. 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 (e.g., 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 HP-IB 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—e.g., 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 HP-IB 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 HP-IB 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 ISRs 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 volume that contains the I/O 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 three 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 making a call to 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 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 linefeed 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 $*xxx*0000 address boundaries 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 $478000.

The code in EXTH is based on the Series 200 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 straightforward 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 HP-IB drivers or you are writing your own HP-IB 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 INT0STAT (note that "0" is a zero, not an uppercase "O") 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 INT0STAT 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 fake isr finishes and returns to the calling code. Any conditions that the fake isr 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 problem where the 9914 can generate an ISR request and then have INT0STAT 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 GPIO 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 *or*ed.

The code in `EXTG` is based on the Series 200 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 HP-IB EOI termination) there are.

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 requesters 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 respect to 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 (98628A/98629A)

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, factory data link (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 powerup. 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 (of 26xx type) connected via an RS-232 interface (the 98626A, 98644 or 98628 interface). If you want to do something other than this sample approach, you need to have familiarity with:

1. `KEYS`, `CRT`, and `SYSDEVS` modules in `INITLIB`,
2. Access methods,
3. I/O drivers and their structure, and
4. `CTABLE`.

Before you do *anything* discussed in this section, be sure you make backup 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 `KEYS` and `CRT` parts of `INITLIB`. This has the advantage of still allowing some interaction on the normal keyboard. The other approach is to replace the `KEYS` and `CRT` 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. It is the approach taken in the following discussion.

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:` disc 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 `DOKBDIO` and `DOCRTIO`.

3. **Install Drivers in** `INITLIB`

   Install these modules in `INITLIB` on your boot disc (via the Librarian). The modules must replace the current modules (i.e., `KEYS` and `CRT`), which must be removed. If your interface driver is not in `INITLIB`, it must also be installed (e.g., RS-232).

4. **Modify** `CTABLE`

   You may wish to change the `TABLE` file on your boot disc 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 disc. (This step is optional with the code given in this chapter.)

Create New Drivers

Each of KEYS and CRT is a separate program and module. The program part takes care of initializing the module. The modules are:

Module	Normally Requires	Purpose
KEYS	SYSDEVS, SYSGLOBALS, ASM, MISC	Support of the keyboard part of the keyboard
CRT	SYSDEVS, SYSGLOBALS, ASM, MISC	Support of the CRT

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 standard driver 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 standard driver, 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 SYSTERM: 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 two 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 two-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 cause this type of reset.

If the interface needs extra software setup (e.g., change baud rate, etc.), this should be done in CTABLE, or just before LAST in INITLIB, as typically the driver for the interface has not run by the time KEYS/CRT run, so there will be no easy way to set up the card. If the card were set up by KEYS/CRT, the interface driver would usually change this as it executed. When CTABLE executes, it usually has drivers at its disposal to facilitate setup. Notice that software setup done this way means that the

```
Loading STARTUP
Loading TABLE
```

messages during booting may be garbled because the interface transmission characteristics do not match terminal reception characteristics. Because of the above, it is best to set up the card via default switches (it if has them), or set the terminal to wake up matching the card.

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 := default_isc;
```

This line will re-direct the I/O to the select code specified by DEFAULT_ISC. 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 disc. For each of the modules that you are replacing (KEYS and CRT), 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:

1.	CREMKEYS ⟨cr⟩ N ⟨cr⟩	Go into the compiler and compile the source REMKEYS (found on your EXAMPL: disc) with no listing and put object code into REMKEYS.CODE.
2.	LOREMKEYS ⟨cr⟩ LIREMKEYS ⟨cr⟩ ALKQ	Go into the librarian and specify an output file of REMKEYS.CODE; link together all the modules of input file REMKEYS.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 the modules with the Librarian. The KEYS and CRT modules are some of the first modules in INITLIB (note that CRTB should be replaced if you are running remote console on a 9837A). When you have replaced 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: disc with a file name of 'INITLIB.'.

Modify Ctable

The CTABLE file can to be modified to allow the remote console to work with a specified interface select code. The normal CTABLE (as shipped with the default system) will specify where the default CONSOLE: and SYSTERM: volumes exist and what type of units they are.

The CTABLE changes to allow for the utilization of the new remote console drivers look like the following. For example, to use select code 21 for the remote console:

```
procedure tea_crt(un:unitnum);
  begin
    tea(un,'MISC_UNBLOCKEDDAM',
        'SYSDEVS_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',
        'SYSDEVS_KBDIO',                 { change drv }
        21,                              { change isc }
        0,0,0,0,0,'SYSTERM',#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. before 'SYSTERM').

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).

The CRT Information Records

Some constants have been set up appropriately in REMCRT for use with remote console. These are CRTFREC, CRTCREC, and CRTIREC. For the example remote console driver in this discussion, some of the ones which have to be changed are:

CRTFREC.HASLCCRT	This boolean variable specifies whether or not there is a "local" CRT. The implication to this is how scrolling takes place: should you just shift an area of memory to the left, or should you call the CRT driver? For the remote console, this is FALSE, indicating that the CRT driver should be called.
CRTIREC.WIDTH and CRTIREC.HEIGHT	This should be the number of columns and rows, respectively, of the display area you wish to use.
CRTIREC.CRTMEMADDR through CRTIREC.CRTCON	These are all unused, because with a remote console, there is no 6845 CRT controller, and no typeahead buffer. They are all set to zero just for cleanliness' sake.

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 (HP-IB) 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 provide functionality similar to that of Stream files.

Problems and Trouble Spots

There are some potential problems with dealing with a remote console. Some of these are:

Area	Problem
Debugger	It is impossible to use from the remote console with the approach given.
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 all preceding 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 with the remote console driver to do remote console (on screen) graphics. However, by modifying the DGL library, it is possible to do remote graphics (HP does not support this).

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 can connect physically but pins 2 and 3 are turned around. It is 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.

5. Default Settings

If the 4.0 Boot ROM tries to remote-boot from a 98644 card (which has no switches to set default datacomm characteristics), the card will always be assigned the following characteristics:

- 8 bits per character,
- No parity,
- A single stop bit,
- 9600 baud, and
- XON/XOFF handshaking.

Pascal defaults for the 98644 card are found in the *Pascal 3.0 Procedure Library* manual, in the *RS-232 Serial* section.

Standard ASCII Keystroke Meanings

Standard keyboard	ASCII	Terminal keyboard
RETURN	CR	RETURN
UP	US	CTRL DEL
DOWN	LF	CTRL J
LEFT	BS	CTRL H
RIGHT	FS	CTRL \
BACKSPACE	BS	BACKSPACE
space bar		space bar
Select	ETX	CTRL C
Shift-Select	ESC	ESC

REMKEYS.TEXT

```
$SYSPROG ON$
$heap_dispose off$
$iocheck off$
$range off$
$ovflcheck off$
$debug off$
$STACKCHECK OFF$


PROGRAM installkeys;

MODULE keys;

IMPORT sysglobals,sysdevs,asm,misc,iodeclarations,general_0,iocomasm;

EXPORT

 PROCEDURE initkeys;

IMPLEMENT

 CONST
  default_isc = 9;

  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 dvrs    }
  newdrivers.iod_init := new_reset;          { put in new reset  }
  isc_table[myisc].io_drv_ptr := ADDR(newdrivers); { install drvs }
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
 WITH isc_table[myisc] DO
  BEGIN
   IF 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 (card_id = hp98626) or (card_id = hp98644) 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; { WITH isc_table[myisc] DO }
END;

PROCEDURE remote_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 := default_isc;
 ioresult := 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
                  ioresult := ORD(ibadrequest);
                END;

 END; { of CASE }
END; { of PROCEDURE }

PROCEDURE dummyreq(cmd : byte; VAR value : byte);
BEGIN END;

PROCEDURE dummykbd(VAR statbyte, databyte : byte;
                   VAR doit : BOOLEAN);
BEGIN END;

PROCEDURE dummyproc;
BEGIN END;

PROCEDURE dummyboolproc(b : BOOLEAN);
BEGIN END;

PROCEDURE initkeys;
VAR localisc  : shortint;
BEGIN
  kbdiohook       := remote_kbdio;
  kbdreqhook      := dummyreq;
  kbdisrhook      := dummykbd;
  kbdpollhook     := dummyboolproc;
  kbdwaithook     := dummyproc;
  kbdreleasehook  := dummyproc;
```

```
  kbdtype         := specialkbd1;
  kbdlang         := ns1_kbd;
  sysmenu         := NIL;
  sysmenushift    := NIL;
  menustate       := m_none;
  FOR localisc := 0 TO 31 DO
   eol_lying_around[localisc] := FALSE;
END;

END;    { of module keys }


IMPORT keys,loader;

BEGIN
  initkeys;
  markuser;
END.
```

REMCRT.TEXT

```
$SYSPROG ON$
$heap_dispose off$
$iocheck off$
$range off$
$ovflcheck off$
$debug off$
$STACKCHECK OFF$

PROGRAM installcrt;

MODULE crt;

IMPORT sysdevs,sysglobals,asm,misc,iodeclarations,general_0 ;

EXPORT

  PROCEDURE crtinit;

IMPLEMENT


CONST dc1          = 17 ;
      default_isc   = 9;

      term_environ=environ[
                miscinfo:crtfrec[
                                  nobreak:FALSE,
                                  stupid :FALSE,
                                  slowterm:FALSE,
                                  hasxycrt:TRUE,
                                  haslccrt:FALSE,
                                  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 :80,height:24,
                                  crtmemaddr:0,
                                  crtcontroladdr:0,
                                  keybufferaddr: 0,
                                  progstateinfoaddr: 0,
                                  keybuffersize: 0,
                                  crtcon: crtconsttype [0,0,0,0,0,0,0,
                                                        0,0,0,0,0],
```

```
                              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]];


VAR   myisc          : shortint;
      newdrivers     : drv_table_type;
      screenwidth        : shortint;
      screenheight       : shortint;
      maxx,maxy          : shortint;

{ 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 dvrs    }
  newdrivers.iod_init := new_reset;           { put in new reset  }
  isc_table[myisc].io_drv_ptr := ADDR(newdrivers); { install drvs }
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 dummyupdatecursor;
BEGIN
END;

PROCEDURE dummydbcrt(op : dbcrtops; VAR dbcrt : dbcinfo);
BEGIN 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(dc1));                        { 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);
  call(updatecursorhook);
END;

PROCEDURE remote_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 := default_isc;
 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
                                    output(CHR(esc) + 'L' + chr(esc) +
                                           '&a0y0C' + chr(esc) + 'K' );
                                    setxy(xpos, ypos);
                                  END;
                                IF (ypos>0)
                                  THEN BEGIN
                                    { out(CHR(esc));
                                    out('A'); }
                                    setxy(xpos,ypos-1);
                                  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(etx): 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; { of IF }
          END;
```

```
                   END; { of CASE c OF }
                   call(updatecursorhook);
                 END; { of WHILE DO BEGIN }
                 IF request = startwrite THEN call(fp^.feot, fp);
               END; { of startwrite, writebytes case }

  OTHERWISE    BEGIN
                 ioresult := ORD(ibadrequest);
               END;

 END; { of CASE request OF }
END;  { of PROCEDURE crtio }

PROCEDURE dummyproc;
BEGIN
  { nothing }
END;

PROCEDURE crtinit;
 BEGIN
   syscom^ := term_environ;
   WITH syscom^.crtinfo DO BEGIN
     screenwidth:=width;
     screenheight:=height;
     maxx       :=width-1;
     maxy       :=height-1;
     xpos       := 0;
     ypos       := 0;
     crtiohook        := remote_crtio;
     dumpalphahook    := dummyproc;
     dumpgraphicshook := dummyproc;
     togglealphahook  := dummyproc;
     togglegraphicshook := dummyproc;
     updatecursorhook := dummyupdatecursor;
     crtinithook      := crtinit;
     crtllhook        := dummycrtll;
     dbcrthook        := dummydbcrt;
     currentcrt       := specialcrt1;
     bitmapaddr       := 0;
     frameaddr        := 0;
     keybuffer^.echo  := FALSE;
     alphastate       := TRUE;
     graphicstate     := FALSE;
   END; { of WITH DO BEGIN }
 END;   { of PROCEDURE crtinit }

END;  { of MODULE crt }

IMPORT crt,loader;

BEGIN
  crtinit;
  markuser;
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 the library IO 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 boot disc 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 which are meant to be included in INITLIB (some were not because of lack of space) and their uses are:

Driver	Use
HPIB	Built-in HP-IB interface and 98624 interfaces. Used by disc and printer drivers that go through these interfaces.
DISC_INTF	For the 98625A high-speed HP-IB disc interface. Standard driver requires the presence of DMA hardware and software.
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 disc 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 remote console (or other unblocked 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 disc drivers require the DMA hardware and this driver to be present, as does the 98625A.
RS-232	For 98626, 98644 and built-in RS-232C interfaces. Only system use is when the table specifies the interface as an unblocked volume (e.g., printer, remote console).

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 into `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    sysglobals , isr , general_0 , extd, iodeclarations ;
EXPORT
  CONST dummy_id    = -100;
        dummy_type  =  100;
  VAR my_dummy_drivers  : drv_table_type;
  PROCEDURE io_init_dummy;
IMPLEMENT

  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:=iominisc 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 keyboard/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 cannot do a `READ/READLN` from the keyboard (since the keyboard 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 a user-interrupt mask has been enabled, 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 ISR service 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 cannot do a READ/READLN from the keyboard (since the keyboard 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 three 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 determine the select code within the ISR 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   { of hpib_5_init  }
  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:

```
$SYSPROG$

PROGRAM isrtest(INPUT,OUTPUT);

$SEARCH 'GPIO5'$      {Or wherever it's hidden}

IMPORT iodeclarations, general_0, general_1, gpio_5;

VAR i, count, dummy: INTEGER;

PROCEDURE myproc(temp: INTEGER);
BEGIN
  writeln('          ISR');
  TRY
    i := ioread_word(15,4);
    writeln('              ',i:6);
    count := count-1;
    if count > 0 then
     iowrite_byte(15,0,0)       {start next handshake}
    else
     off_flag(15);              {finished, so turn off interrupt}
  RECOVER
   writeln('            ISR ESCAPE!');
END; {PROCEDURE myproc}

BEGIN
  i     := -1;                  {set i to known value}
  count := 10;                  {input 10 data words from GPIO card}
  set_timeout(15,1.0);
  dummy := ioread_word(15,4);    {set I/O direction to "input"}
  iowrite_byte(15,0,0);          {start first handshake with peripheral}
  on_flag(15,myproc,0);
  WHILE count > 0 DO
   WRITELN('Waiting....',i:6)
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,BINCMP(flgmask));
      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  }
      iocontrol(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   { of gpio_5_init  }
  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.
```

Level 5 routines have been written for neither the 98626A nor the 98644A.

```
$COPYRIGHT 'COPYRIGHT (C) 1982 BY HEWLETT-PACKARD COMPANY'$
$SYSPROG ON$
$PARTIAL_EVAL ON$
$STACKCHECK ON$
$RANGE OFF$
$DEBUG ON$
$OVFLCHECK OFF$
(*************************************************************)
(*                                                           *)
(*      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,BINCMP(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);
        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,BINCMP(data_mask ));
      prmpt_cond:  mask:=BINAND(mask,BINCMP(prmpt_mask));
      fperr_cond:  mask:=BINAND(mask,BINCMP(fperr_mask));
      mdmch_cond:  mask:=BINAND(mask,BINCMP(mdmch_mask));
      noact_cond:  mask:=BINAND(mask,BINCMP(noact_mask));
      lstcr_cond:  mask:=BINAND(mask,BINCMP(lstcr_mask));
      eol_cond:    mask:=BINAND(mask,BINCMP(eol_mask  ));
      break_cond:  mask:=BINAND(mask,BINCMP(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,prmpt_cond);
```

```
END;
PROCEDURE off_prompt (isc       : type_isc);
BEGIN
  serial_isr_kill(isc,prmpt_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   { of serial_5_init  }
  FOR counter := iominisc TO iomaxisc DO
    serial_isr_table[counter] := NIL;
END.    { of serial_5_init  }
```

Index

a

$A804XDVR$ module 32, 221
Absolute image 21
Absolute modules 13
Absolute start address 13
Access methods (see AMs) 36
Address Error exception 19
Address lines 18
Allocation of variables 11
Alpha displays 266
Alpha screen drivers 275, 278
Alpha screen dumping 291
ALPHASTATE 291
AMREQUESTTYPE 38
AMs (Access Methods):
- AMREQUESTTYPE 38
- AMs 36, 38, 42, 48
- AMTABLE 43
- AMTYPE 40
- ASCIIAM 39
- SERIALTEXTAM 39
- SRMAM 39
- STANDARDAM 39
- TEXTAM 39
- UNBUFFEREDAM 39

AMTABLE 43
AMTYPE 40
ANYCHAR key 222
ANYTOMEM procedure 177
APPEND 47
ASCII 36
ASCII file 35
ASCIIAM 39
ASM module 32
ASM_CLOSEFILES 68
Assembler 13
Assembly Language 2
Attribute byte 278
Auto-configuration 7
Auto-repeat 242

b

BA 53
BADFILE 36
BADIO procedure 176
BASIC 15
BATTERY 333, 343
BATTERY variable 365
Beep compatibility 236
Beeper 217, 230, 245
Beeper (4-voice) 233
Bit-mapped display 282
Bit-mapped displays 276
"Black-Box" description 237
BLDS (byte lower data strobe) 18
BLFLAGS variable 365
Boot 35
BOOT: disc 15, 357
Boot disc formats 316
Boot file 21
Boot formats 312
Boot ROM 15, 309
Boot ROM configuration 339
Boot ROM revision 339
BOOTTYPE variable 343, 365
BottomBytesStolen 20
BUDS (byte upper data strobe) 18
Bus error 19, 267, 275, 279
Byte address 53
BYTEOFFSET 54

c

Cache memory 8, 9
Calendar 218
CAT procedure 194
CATALOG 156, 171
CATENTRY 160
Centiseconds 218
CHANGE procedure 196
CHANGEFNAME procedure 167
CHANGENAME 155, 170
CHANGEPASSWORD procedure 200
Character generation 279
Character sets 266

Character table 361
CHARLIE module 30
Checksum 251, 315
CI (Command Interpreter) 26, 28
CLEANDIR procedure 164
Clock 218, 229
CLOSEDIR procedure 189
CLOSEDIRECTORY 157, 171
CLOSEF procedure 166
CLOSEFILE 153, 169
CLOSEINFILE procedure 182
CLOSEOUTFILE procedure 183
CMD_READ_1 .... 237, 252
.CODE 36
Code modules 13
Color map 273
Command interpreter 16, 26
Command register (floppy controller) 298
Command Set 80 disc drives 9
Compiler 13
Compiler Directives:
  $DEBUG$ 60, 210
  $IOCHECK$ 11, 66, 69
  $MODCAL$ 10
  $OVFLCHECK$ 210
  $RANGE$ 11, 210
  $STACKCHECK$ 11, 210
  $SYSPROG$ 10
  $TABLE$ 11
Configuration (of Boot ROM) 339
Configuration jumper 239
Configuration Mode software override 341
Configuration of machine 332
Configuration program 16
CONSOLE: 289
Control registers (4-voice beeper) 234
Control registers (floppy controller) .. 296
Controller board RAM (floppy drives) 299
CPU interrupt handling 203
CPU state at load 343
CRASH 343, 361
CREATEDIR procedure 189
CREATEFILE 152, 168
CRT 6
CRT controller 279
CRT interrogation 291
CRT module 32
CRT presence 335
CRT RAM test (hi-res) 341
CRT registers 10 and 11 279
CRT registers 12 and 13 281
CRT registers 14 and 15 281
CRTB module 32
CRTBLANK 367
CRTCLEAR 360, 367
CRTID 335, 337
CRTINIT 359, 367
CRTMSG 360, 367
CRUNCH 154, 171
CRUNCHV procedure 164
CS80 disc drives 9
CTABLE program 7
Cursor control 279
Cursor movement 289
CVTDATETIME procedure 164
Cycle timer 238
Cyclic interrupt 244

d

DAMREQUESTTYPE 41
DAMs (Directory Access Methods):
  DAMREQUESTTYPE 41
  DAMs 16, 41, 53, 149
  Golden Rule for DAMs 150
  INSTALLLIFDAM 158
  INSTLIFDAM program 158
  LIFDAM procedure 162
DATA 36
Data encoding 294
Data register (floppy controller) 299
Data requests from within an ISR ... 232
Data transfers 294
Data Transfers (68xxx-HP-HIL) 259
Date 243
$DEBUG$ compiler directive 60, 210
Decoding data for floppies 294
Default mass storage 325
DEFAULT_MSUS 325, 343
DEFs 31
Delay (auto-repeat) 242
Delay timer 238
Delayed interrupt 244
Dependencies, module 33
Destination pattern 282
Destructive read 207
Device configuration identification ... 334
Device I/O 35
Device ID 54

Device Type byte ... 325
DEVID ... 54
DGL ... 286
Differences Among Pascal Releases ... 6
Directory:
  CLEANDIR procedure ... 164
  CLOSEDIR procedure ... 189
  CLOSEDIRECTORY ... 157, 171
  CREATEDIR procedure ... 189
  Directory Access Methods (see DAMs) ... 41
  DOMAKEDIRECTORY procedure ... 164
  DOOPENDIRECTORY procedure ... 164
  FLUSHDIR procedure ... 164
  IDIRFULL ... 70
  IDIRNOTEMPTY ... 70
  IFILENOTDIR ... 71
  INEEDTEMPDIR ... 71
  INODIRECTORY ... 70
  INOTONDIR ... 71
  MAKEDIR procedure ... 190
  MAKEDIRECTORY ... 155
  OPENDIR procedure ... 162, 188
  OPENDIRECTORY ... 155, 171
  OPENPARENTDIRECTORY ... 155
  READDIR ... 60
Disassembly ... 61
Disc drives (internal) ... 293
Disc Unit ... 54
Disc volume ... 54
Discs:
  Boot disc formats ... 316
  BOOT: ... 15
  Command Set 80 drives ... 9
  CS80 drives ... 9
  Drives (internal) ... 293
  EPROM pseudo-disc format ... 324
  Flexible disc drivers ... 350
  Flexible disc drives ... 293
  HP7905 hard disc ... 325, 326
  HP7906 hard disc ... 325, 326, 327
  HP7908 hard disc ... 9
  HP7920 hard disc ... 325, 326
  HP7925 hard disc ... 325, 326
  HP82900 series flexible disc ... 326
  HP8290X disc ... 9
  HP9122 disc ... 9, 327
  HP913X A winchester disc ... 326
  HP913X B winchester disc ... 326
  HP913X C winchester disc ... 326
  HP913X D winchester disc ... 9
  HP98255 EPROM pseudo-disc ... 326
  HP9885 flexible disc ... 326
  HP9895 flexible disc ... 326
  Internal ... 293
  ROM/EPROM pseudo-disc format 324
  SS80 discs ... 9
  Subset 80 discs ... 9
  Unit ... 54
  Volume ... 54
Display control block ... 284
Display hardware capabilities ... 267
Displays ... 266
DISPOSE ... 68
DOCAT procedure ... 164
DOMAKEDIRECTORY procedure ... 164
DOOPENDIRECTORY procedure ... 164
DOOVERWRITEFILE procedure ... 167
DOPREFIX ... 75
DOPURGENAME procedure ... 164
DRIVE variable ... 365
Drivers:
  Alpha screen ... 275, 278
  Flexible disc ... 350
  Graphics screen ... 287
  I/O ... 15
DRVR_KEY variable ... 364
DTACK error ... 267
DU ... 54
Dummy modules ... 34
DUMPALPHAHOOK ... 291
DUMPGRAPHICHOOK ... 291
DUPLICATE procedure ... 195
DV ... 54
DVRTEMP ... 54
Dynamic linking ... 14

e

Echoing read ... 215
EFTTABLE ... 43
Encoding data for floppies ... 294
ENDCAT procedure ... 191
ENDIOERRS ... 71
ENDPASS procedure ... 197
Entry Points ... 8
EPROM pseudo-disc format ... 324
ERROR message ... 365
ESCAPE ... 69

ESCAPECODE 204, 210
Exception vectors 362
Exceptions 19
EXTEND CHAR key 221
Extended command 296
Extended Status 297
Extended status clear 298

f

FACCESS 67
FANONCTR 50
FANONFILE 76
FANONYMOUS 46
Fast handshake time 238
Fast-blinking cursor 279
FB0, FB1 51
FBLOCKIO 37, 78
FBUFCHANGED 47
FBUFFER 51
FBUFFERED 46
FBUFFERREF 37, 80
FBUFVALID 46
FBUSY 50
FCLOSE 81
FCLOSEIT 37, 82
FEFT 50
FEOF 37, 47, 83
FEOLN 37, 47, 84
FEOT 50
FEXTRA 51
FEXTRA2 51
FFPW 50
FGET 37, 85
FGETXY 86
FGOTOXY 37, 87
FHPOPEN 37, 88
FHPRESET 37, 91
FIB 45, 290
FIBs (File Information Blocks):
  FIB 45, 290
  Fields of a FIB 45
  FINISHFIB procedure 166
  SETUPFIBFORFILE procedure 180
Fields of a FIB 45
File I/O 35
File system 35, 289
File type 36
File variables 36
FILECOPY procedure 183
FILEDEMO program 60
FILEID 49
FILEKIND 38
FILEPACK:
  Cataloging a directory .. 173, 188, 192
  Changing a file name 173, 196
  Changing file passwords ... 173, 200
  Copying a file 173, 183
  Creating a volume directory .. 173, 189
  Duplicating a link 173, 195
  Listing file passwords 173, 197
  Listing the volumes on line ... 173, 187
  Making a directory 173
  Making a file 173, 191
  Removing a file 173, 196
  Repacking a volume 173, 188
  Setting default prefixes .. 173, 201
  Setting unit prefixes 173, 201
  Translating a file 173, 183
Files:
  ASCII 35
  ASM_CLOSEFILES 68
  BADFILE 36
  Boot file 21
  CLOSEFILE 153, 169
  CLOSEINFILE procedure 182
  CLOSEOUTFILE procedure 183
  CREATEFILE 152, 168
  DOOVERWRITEFILE procedure 167
  FANONFILE 76
  File I/O 35
  File system 35, 289
  File type 36
  File variables 36
  FILECOPY procedure 183
  FILEDEMO program 60
  FILEID 49
  FILEKIND 38
  FINDFILE procedure 164
  FOVERFILE 99
  IBADFILETYPE 70
  IDUPFILE 69
  IFILELOCKED 70
  IFILENOTDIR 71
  IFILEUNLOCKED 70
  ILOSTFILE 69
  INITLIB file 15
  INOFILE 69
  LAST file 16

LIF system file format 318
MAKEFILE procedure 191
OPENFILE 151, 167
OPENing a file 47
OUTPUT file 289
OVERWRITEFILE 168
PURGEFILE 169
RESETting a file 47
SETUPFIBFORFILE procedure 180
SRM system files 324
Stream files 28
System files 17
SYSTEM_P 15
TABLE 16
UNTYPED file 39
UNTYPEDFILE 36
Workfiles 28
FINDFILE procedure 164
FINDVOLUME 93
FINISHFIB procedure 166
FINITB 94
FINTRUPT 354
FISNEW 46
FISTEXTVAR 45
FIXNAME 95
FKIND 36, 45
FLASTPOS 48
FLEOF 48
Flexible disc drivers 350
Flexible disc drives 293
FLISTPTR 45, 65
Floating RAM 18, 21
Floating-point arithmetic 15
FLOCKABLE 50
FLOCKED 51
Floppy controller board RAM 299
Floppy controller chip 293
Floppy Controller:
Command register 298
Control registers 296
Data register 299
Floppy controller board RAM 299
Floppy controller chip 293
Force Interrupt command 306
Interrupt 306
On-board RAM 299
RAM 299
Read Address command 305
Read Sector command 304
Read Track command 305
Restore command 303
Sector register 299
Seek command 303
Status information 306
Status register 298
Status registers 296
Step command 303
Step-in command 303
Step-out command 303
Track register 299
Type I commands 300, 303
Type II commands 301, 304
Type III commands 302, 305
Type IV commands 302, 306
Write Sector command 304
Write Track command 305
FLPYINIT 354
FLPYMREAD 355
FLPYMWRITE 355
FLPYREAD 353
FLPY_WRITE 353
FLUSHDIR procedure 164
FM (frequency modulation) 294
FMAKETYPE 96
FMAXPOS 37, 98
FMODIFIED 47
FMSGS 356
FOPTSTRING 50
Force Interrupt command
(floppy controller) 306
Four-voice beeper 233, 245
FOVERFILE 99
FOVERPRINT 37, 101
FOVERWRITTEN 50
FPAGE 37, 102
FPEOF 48
FPOS 47
FPOSITION 37, 103
FPURGEOLDLINK 50
FPUT 37, 104
Frame buffer 276
Frame buffer allocation 285
FREAD 37, 105
FREADABLE 46
FREADBOOL 37, 106
FREADBYTES 107
FREADCHAR 37, 108
FREADENUM 37, 109
FREADINT 37, 110
FREADLN 37, 111

FREADMODE . . . 46
FREADPAOC . . . 37, 112
FREADREAL . . . 37, 113
FREADSTR . . . 37, 114
FREADSTRBOOL . . . 115
FREADSTRCHAR . . . 116
FREADSTRENUM . . . 117
FREADSTRINT . . . 118
FREADSTRPAOC . . . 119
FREADSTRREAL . . . 120
FREADSTRSTR . . . 121
FREADSTRWORD . . . 122
FREAWORD . . . 123
FRECSIZE . . . 45
FREPTCNT . . . 48
FS module . . . 33
FSEEK . . . 37, 124
FSTARTADDRESS . . . 48
FSTRIPNAME . . . 125
FS_FCLOSEIT . . . 66
FS_FHPOPEN . . . 66
FS_FINITB . . . 65
FS_FWRITE . . . 66
FS_FWRITEPAOC . . . 67
FTID . . . 49
FUBUFFER variable . . . 365
FUNIT . . . 50
FVID . . . 49
FWINDOW . . . 45
FWRITE . . . 37, 126
FWRITEABLE . . . 46
FWRITEBOOL . . . 37, 127
FWRITEBYTES . . . 128
FWRITECHAR . . . 37, 129
FWRITEENUM . . . 37, 130
FWRITEINT . . . 38, 131
FWRITELN . . . 38, 132
FWRITEPAOC . . . 38, 133
FWRITEREAL . . . 38, 134
FWRITESTR . . . 38, 135
FWRITESTRBOOL . . . 136
FWRITESTRCHAR . . . 137
FWRITESTRENUM . . . 138
FWRITESTRINT . . . 139
FWRITESTRPAOC . . . 140
FWRITESTRREAL . . . 141
FWRITESTRSTR . . . 142
FWRITESTRWORD . . . 143
FWRITEWORD . . . 38, 144
F_AREA . . . 343
F_AREA variable . . . 364
F_PWR_ON . . . 352

g

Gbase . . . 65
GET . . . 35
GETSPACE function . . . 165
GETSYSDATE procedure . . . 164
GETVOLUMEDATE . . . 154, 170
GETVOLUMENAME . . . 154, 170
Global area . . . 23, 26
Global constants . . . 8
Global variable area . . . 22
Golden Rule for DAMs . . . 150
Graphics presence . . . 335
Graphics screen drivers . . . 287
Graphics screen dumping . . . 291
GRAPHICSTATE . . . 291

h

Handshake . . . 231
Handshake time . . . 238
Hanging reads . . . 217
Hard system . . . 15
Hardware (display) . . . 267
Heap . . . 22
Hi-res CRT RAM test . . . 341
High memory . . . 19
High RAM map . . . 362
Highlight byte . . . 278
Highlighting . . . 289
HP-HIL . . . 211, 233, 240, 245
HP-HIL interface . . . 253
HP46020A keyboard . . . 211, 227
HP7905 disc . . . 325
HP7906 disc . . . 325, 327
HP7908 disc . . . 9
HP7920 disc . . . 325
HP7925 disc . . . 325
HP82900 series flexible disc . . . 326
HP8290X disc drives . . . 9
HP9122 disc . . . 327
HP9122 disc drive . . . 9
HP913X A winchester disc . . . 326
HP913X B winchester disc . . . 326
HP913X C winchester disc . . . 326

HP913XD disc drives 9
HP98203A keyboard . . 211, 225, 228, 229
HP98203B keyboard 211, 226, 229
HP98204A board 287
HP98204B board 287
HP98255 EPROM pseudo-disc 326
HP98259 Bubble memory card 326
HP98635A card 4
HP9885 flexible disc 326
HP9895 flexible disc 326
HPL 15, 320
Human Interface 6

i

I/O drivers 15
I/O mapping addresses 206
IBADCLOSE 70
IBADFILETYPE 70
IBADFORMAT 69
IBADPASS 70
IBADREQUEST 70
IBADTITLE 69
IBADUNIT 69
IBADVALUE 70
IBF (input buffer full) flag 249
ICANTSTRETCH 70
ID PROM 329
IDIRFULL 70
IDIRNOTEMPTY 70
IDUPFILE 69
IEOF 70
IFILELOCKED 70
IFILENOTDIR 71
IFILEUNLOCKED 70
ILOSTFILE 69
ILOSTUNIT 69
Induction Rule 14
INEEDTEMPDIR 71
Initialization:
  Module initialization 31
  ZUNINITIALIZED 70
INITLIB file 15
INITLIB program 22, 24, 26, 28
INITLOAD module 32
INITUNITS module 33
INOACCESS 70
INODIRECTORY 70
INOERROR 69
INOFILE 69
INOROOM 69
INOTCLOSED 69
INOTDIRECT 70
INOTLOCKABLE 70
INOTONDIR 71
INOTOPEN 69
INOTREADABLE 70
INOTVALIDSIZE 70
INOTWRITEABLE 70
INOUNIT 69
Input Buffer Full flag 249
INSTALLLIFDAM 158
INSTLIFDAM program 158
Interface cards 16
Interface text 34
Interface to HP-HIL 253
Internal disc drives 293
Internal Peripherals 6
Interrogating the CRT 291
INTERRUPT 204, 210
Interrupt (floppy controller) 306
Interrupt levels 362
Interrupt mask 239
Interrupt routines 11
Interrupting the CPU 230
Interrupts 19
Interrupts:
  CPU interrupt handling 203
  Cyclic interrupt 244
  Delayed interrupt 244
  Masking interrupts 245
  Phantom interrupt 207
  Request interrupt mask 239
  Soft interrupt vectors 362
INTERRUPTTABLE 206
$IOCHECK$ compiler directive . . 11, 66, 69
IOCHECK procedure 175
IOERRMSG function 175
IORESULT 39, 69, 204, 210
ISR module 32
ISRIB (ISR Information Block) 205
ISRLINK 206
ISRMCATCHALL 71
ISRs 204
ISRs (Interrupt Svc. Routine):
  Data requests from within an ISR . . 232
  ISR module 32
  ISRIB (ISR Information Block) . . . 205
  ISRLINK 206

ISRs 204
ISRUNLINK 206
KBDISRHOOK data byte 224
KBDISRHOOK procedure 221
KBDISRHOOK status byte 223
PERMISRLINK 209
RPGISRHOOK data byte 224
RPGISRHOOK procedure 221
RPGISRHOOK status byte 223
TIMERISRHOOK procedure 221
ISRUNLINK 206
ISTROVFL 70
ITOOMANYOPEN 70

k

KBDISRHOOK data byte 224
KBDISRHOOK procedure 221
KBDISRHOOK status byte 223
KBDSADR 214, 249, 257, 259
KBDSYSMODE 222
KBDTRANSHOOK procedure 221
Kernel 13, 15, 23, 29
Keyboard 338
Keyboard at power-up 251
Keyboard at reset 251
Keyboard controller 6
Keyboard cooking 214
Keyboard handling protocol 230
Keyboard processor 229
Keyboard remapping 219
Keyboards 211
Keymap 261
KEYS module 32
Knob (see RPG) 213
Knob pulse period 242

l

Language code 240
LAST file 16
LAST program 26
LETTER 55
Librarian 13, 21, 61
Libraries 13
LIBRARY 14
LIF 35, 293, 326
LIF system file format 318
LIF:
INSTALLLIFDAM 158
INSTLIFDAM program 158
LIF system file format 318
LIFDAM 162
LIFMODULE module 160
LIFVOL function 162
the directory structure ... 35, 293, 326
LIFDAM procedure 162
LIFMODULE module 160
LIFVOL function 162
Linked object code 21
Linking 13
LISTATTRIBUTE procedure 198
LISTPASSWORD procedure 199
LOADER module 32
LOADER program 22
Loading 13
LOCK 66
LOCKDOWN 159
LOCKUP 159
Long memory test 341
Loopback mode 253
Low ROM map exception vectors 366
LOWRAM variable 365
LPCTRL 249, 257
LPSTAT 249, 257, 258

m

Machine configuration 332
Major revisions 8
MAKEDIR procedure 190
MAKEDIRECTORY 155
MAKEFILE procedure 191
MAKELINK 157
Manuals:
MC68000 User's Manual .. 5, 343, 360
Pascal 3.0 Graphics Techniques ... 382
Pascal 3.0 Procedure
Library ..5, 203, 228, 245, 382, 465, 496
Pascal 3.0 Workstation
System 3, 5, 6, 7, 10, 16, 309, 324
Mapping the keyboard 219
Mask interrupts 245
MASS STORAGE IS 325
Match time 238, 243
MC68000 User's Manual 5, 343, 360

Memory allocation 11
Memory board switches 19
Memory boards 19
Memory map 18, 21
Memory test length 340, 341
Memory wasting 287
MEMTOANY procedure 177
MFM (modified frequency modulation) 294
MIN function 175
MINI module 33
Minifloppy drives 15
Minor revisions 8
MISC module 33, 160
$MODCAL$ compiler directive 10
Model 216 . . . . 267, 287
Model 217 . . . . 268, 287
Model 220 . . . . 269, 287
Model 226 . . . . 271, 287
Model 236A 272, 287
Model 236C 273, 287
Model 237 . . . . 277, 287
Module CHARLIE 30
Module dependencies 33
Module initialization 31
Modules:
- A804XDVR 32, 221
- Absolute 13
- ASM 32
- CHARLIE 30
- Code 13
- CRT 32
- CRTB 32
- Dependencies 33
- Dummy 34
- FS 33
- Initialization 31
- INITLOAD 32
- INITUNITS 33
- ISR 32
- KEYS 32
- LIFMODULE 160
- LOADER 32
- MINI 33
- MISC 33, 160
- Modules 13
- NONUSKBD1 32
- NONUSKBD2 32
- Object code 13
- Pascal 29
- Permanently loaded 25
- Relocatable 13
- SETUPSYS 33
- SUE 30
- SYSDEVS 9, 33, 218, 286
- SYSGLOBALS 32, 40

Mother board 229
msi 325
M_FCLOSE 349
M_FOPEN 347
M_INIT 347
M_READ 349

n

NDRIVES 338, 343
NDRIVES variable 364
NEC (Nippon Electric Company)
monitor 269
NMI (Non-Maskable Interrupt):
- NMI . . 203, 220, 230, 237, 243, 249, 360
- NMI_DECODE 360
- Non-maskable timeout 243

NMI_DECODE 360
Noise generator 235
Non-echoing read 216
Non-maskable timeout 243
Non-present RAM 19
Non-present ROM 19
NONUSKBD1 module 32
NONUSKBD2 module 32
NOWOPEN procedure 167

o

Object Code Incompatibility 8
Object code modules 13
OFFLINE 56
"Old" character set 266
On-board RAM (floppy controller) . . 299
OPENDIR procedure 162, 188
OPENDIRECTORY 155, 171
OPENF procedure 166
OPENFILE 151, 167
OPENing a file 47
OPENNEW procedure 166
OPENOLD procedure 166
OPENPARENTDIRECTORY 155
OPENUNIT 171

OPENVOLUME 157, 171
OUTPUT 67
Output buffer (timer) 241
OUTPUT file 289
OVERWRITEFILE 168
$OVFLCHECK$ compiler directive 210

p

PAC (Packed Array of Characters) .. 215
Packed Array of Characters (PAC) .. 215
PAD 57
Pascal 3.0 Graphics Techniques manual 382
Pascal 3.0 Procedure Library manual ...5, 203, 228, 245, 382, 465, 496
Pascal 3.0 Workstation System manual 3, 5, 6, 7, 10, 16, 309, 324
Pascal kernel 29
Pascal modules 29
PATHID 49
Peripheral configuration 16
Peripheral devices 16
Peripheral support 9
Permanently loaded modules 25
PERMISRLINK 209
Phantom interrupt 207
Power-up options 340
POWERUP 204
POWERUP program 22
PREFIX procedure 201
Prerequisites 5
Primary Address byte 325, 327
Processing an 804x service request ... 231
Program counter 15
Programs:
  Configuration program 16
  CTABLE 7
  FILEDEMO 60
  INITLIB 22, 24, 26, 28
  INSTLIFDAM 158
  LAST 26
  LOADER 22
  POWERUP 22
  Program counter 15
  STARTUP 16, 26
  Stepping through a program 11
  TABLE 26
Protocol for keyboard handling 230
Pulse period (knob) 242
PURGE 68
PURGEF procedure 164
PURGEFILE 169
PURGENAME 155, 170
PUT 35

r

RAM (floppy controller board) 299
RAM (Random-access Memory) 18
RAM board switches 21
RAM data (804x) 239
RAM map 362
RAM pointer (804x) 244
$RANGE$ compiler directive 11, 210
Rate (auto-repeat) 242
Raw data 214
Rbase 65
Read Address command (floppy controller) 305
Read interface 344
Read language code 240
Read Sector command 304
Read timer output buffer 241
Read Track command (floppy controller) 305
Read-Only Memory (ROM) 15
READDIR 60
Real-time match 243
Registers (804x) 244
Relocatable modules 13
Relocatable object code 21
Remapping the keyboard 219
REMOVE procedure 196
REPACK procedure 188
Repeat delay 242
Repeat rate 242
Replacement rule 282
Request 804x RAM data 239
Request 804x register 241
Request configuration jumper 239
Request interrupt mask 239
REQ_BOOT 357
REQ_REBOOT 357
RESET 35, 352
Reset 804x RAM pointer 244
[RESET] key 229, 249
RESETting a file 47

Resetting the 804x 250
Restore command (floppy controller) . 303
Resultant bit 282
RETRY variable 364
REVID 257, 259
Revision (of boot ROM) 339
REWRITE 35, 66
ROM 13
ROM (boot) 309
ROM (character generation) 279
ROM (Read-Only Memory) 15
ROM headers 312
ROM/EPROM pseudo-disc format .. 324
Roman 8 character set 266
Roman Extension characters 266
Rotary Pulse Generator (knob) 213
RPG (Rotary Pulse Generator):
  Knob pulse period 242
  Pulse period (knob) 242
  Rotary Pulse Generator (knob) ... 213
  RPG (knob) 213, 229
  RPGISRHOOK data byte 224
  RPGISRHOOK procedure 221
  RPGISRHOOK status byte 223

S

SAMEDEVICE function 176
SC 53
SCANTITLE 145
Scrolling 290
SDF 326
SDF boot are format 322
Secondary Loading 344
Sector register (floppy controller) 299
SEEK 47
Seek command (floppy controller) ... 303
Select code 53
Select Code byte 325, 327
Selecting an Access Method 42
Self-test looping 340
SENDCMD 237, 252
SENDDATA 237, 252
Sending a command to the 804x 231
SERIALTEXTAM 39
SETSYSDATE procedure 164
SETUNITPREFIX 157, 171
SETUPFIBFORFILE procedure 180
SETUPSYS module 33
SETVOLUMEDATE 154, 170
SETVOLUMENAME 154, 170
Shared Resource Management (see SRM) 9
Slow-blinking cursor 279
Soft interrupt vectors 362
Soft system 15, 21
SOFT SYSTEM ONLY message 365
Softkey area 281
Sound generator 233
Source bit 282
Source code compatibility 8
SRM (Shared Resource
Management) 9, 15, 17, 35
SRM system files 324
SRMAM 39
SS80 disc drives 9
SSP (Supervisor Stack Pointer) ...9, 203
Stack 22
Stack frame 22
Stack frame pointer 24
Stack overflow 22
Stack pointer 15, 346
$STACKCHECK$ compiler directive ..11, 210
STANDARDAM 39
Start address 13
STARTADDRESS variable 365
STARTCAT procedure 192
STARTLISTPASS procedure 197
STARTUP program 16, 26
Status code 5X 255
Status code 6X 255
Status information (floppy controller) 306
Status register 229, 237
Status register (CRT) 283
Status register (floppy controller) 296, 298
STATUS5HOOK 214
STATUS5HOOK procedure 221
STATUS6HOOK 214
STATUS6HOOK procedure 221
Step command (floppy controller) ... 303
Step-in command (floppy controller) .. 303
Step-out command (floppy controller) 303
Stepping through a program 11
STOP key 204
Stream files 28
STRETCHF procedure 167
STRETCHIT 153, 170
Subset 80 disc drives 9
SUE module 30
SUFFIX 147

Suffix compression ... 161
Suffixes ... 42, 161
SUFFIXTABLE ... 43
Supervisor mode ... 19, 21
Supervisor stack ... 22
Supervisor Stack Pointer (SSP) ... 203
Supervisor state ... 9
Switches, memory board ... 19
Symbol table ... 23, 34
SYS ... 36
SYSCLOCK function ... 218
SYSDEFS table ... 23
SYSDEVS module ... 9, 33, 218, 286
SYSFLAG ... 267, 332, 335, 343, 365
SYSFLAG2 ... 333, 343, 364
SYSGLOBALS module ... 32, 40, 67
SYSIORESULT ... 69
SYSNAME variable ... 365
$SYSPROG$ compiler directive ... 10
System clock ... 218
System distribution ... 7
System files ... 17
System library ... 24
SYSTEM NOT FOUND message ... 365
System stack ... 28
System switching ... 357
System volume ... 16
Systems:
  Distribution ... 7
  File ... 35, 289
  Files ... 17
  Hard ... 15
  LIF system file format ... 318
  Soft ... 15, 21
  SRM system files ... 324
  Switching ... 357
  System volume ... 16
  SYSTEM_P file ... 15

t

$TABLE$ compiler directive ... 11
TABLE file ... 16
TABLE program ... 26
Texas Instruments 9914 HP-IB chip .. 207
Texas Instruments SN76494 sound generator ... 233, 259
.TEXT ... 36
TEXTAM ... 39
TI SN76494 sound generator ... 233, 259
TI9914 HP-IB chip ... 207
Time of day ... 238, 243
Timer output buffer ... 241
TIMERISRHOOK procedure ... 221
Timing ... 218
TMs (Transfer Methods):
  TMs ... 16, 40, 53
  TMTYPE ... 40
TOGGLEALPHAHOOK ... 291
TOGGLEGRAPHICSHOOK ... 291
Track register (floppy controller) ... 299
Transfer Methods (see TMs) .. 16, 40, 53
TRAP instruction ... 60, 66, 363
Trick ... 34
Trigger commands ... 245
Trigger data ... 245
TRY/RECOVER ... 22, 68, 69, 204, 209, 210
Turning Screens On and Off ... 291
Type byte ... 326
Type I commands (floppy controller) ... 300, 303
Type II commands (floppy controller) ... 301, 304
Type III commands (floppy controller) ... 302, 305
Type IV commands (floppy controller) ... 302, 306

u

UCSD Pascal ... 44
UISBLKD ... 57
UISFIXED ... 57
UISINTERACTIVE ... 56
UMAXBYTES ... 58
UMEDIAVALID ... 56
Unblocked stream of bytes ... 35
UNBUFFEREDAM ... 39
Uncooked keyboard ... 214
Unit byte ... 325, 327
Unit table ... 40, 52
Unit volume ID ... 54
UNITABLE ... 16, 40, 49, 52
UNITABLETYPE ... 40
UNITBUSY ... 217
UNITENTRY ... 40
UNITNUMBER function ... 176
UNIX ... 15, 39, 316, 326, 344

UNIX boot area format 323
UNTYPED file 39
UNTYPEDFILE 36
UREPORTCHANGES 57
User mode 19, 21
User Stack Pointer (USP) 203
User state 9
USP (User Stack Pointer) 9, 203
UUPPERCASE 57
UVID 54

V

Variable allocation 11
VOLUMES procedure 187

W

Wasting memory 287
Western Digital Corp. FD179X 293
Window width register 283
Workfiles 28
Write Sector command
(floppy controller) 304
Write to 804x register 244
Write Track command
(floppy controller) 305
WRITELN 67
WS1.0 35

Z

ZAPSPACES 148
ZBADBLOCK 69
ZBADDMA 70
ZBADHARDWARE 70
ZBADMODE 69
ZCATCHALL 70
ZINITFAIL 70
ZMEDIUMCHANGED 71
ZNOBLOCK 70
ZNODEVICE 70
ZNOMEDIUM 70
ZNOSUCHBLK 69
ZNOTREADY 70
ZPROTECTED 70
ZSTRANGEI 70
ZTIMEOUT 69
ZUNINITIALIZED 70

Part No. 98615-90074
E 0285
Microfiche No. 98615-99074

Printed in U.S.A.
Edition 1
February 1985